SERIE
AUTOAPRENDIZAJE
ACELERADO



BASIC para electrónicos

Mike James

Aplicaciones prácticas al cálculo de circuitos

La electrónica como software

SPECTRUM... COMMODORE... AMSTRAD... DRAGON... ORIC... APPLE... IBM-PC



BASIC para electrónicos

MIKE JAMES

BASIC para electrónicos

EDICIONES TECNICAS REDE, S.A.
Ecuador, 91 - Tlfno. 250.30.97
08029 BARCELONA

Título original: BASIC Programming for Electronics

Traducción: José M.ª Rodelgo

ISBN: 84-247-0227-1

Impreso en España — Printed in Spain

Dep. Legal: B. 4755-86 — REDEPRINT — Barcelona

SUMARIO

PROLOGO

Hubo un tiempo en que el interés en los microordenadores partía de un buen conocimiento de electrónica. Al fin y al cabo, el que quería utilizar un microordenador tenía que construírselo. Hoy en día, sin embargo, el usuario de un microordenador puede salir adelante sin saber lo más mínimo sobre los componentes que constituyen su equipo, y probablemente ni siquiera tendrá un soldador, ni mucho menos sabrá usarlo. La disponibilidad comercial de los microordenadores ya montados ha dado lugar a una verdadera separación entre el soporte lógico (software) y el soporte físico (hardware) de la informática. Y así como muchos programadores no se molestan en adquirir ningún conocimiento de electrónica, porque no lo necesitan, también mucha gente atraída por el funcionamiento interno del microordenador no se molesta en aprender a utilizarlo. En otras palabras si usted no ha tenido que construir su primer microordenador, no vale la pena que intente aprender electrónica.

Ambas posiciones muestran una verdadera cortedad de visión. Para mí, la verdadera diversión que puede ofrecer la informática está en la combinación del aspecto software con el aspecto hardware. En otras palabras, lo que me interesa es poner en práctica mis conocimientos de electrónica y saber lo que está pasando en el interior del microordenador para mejorar mis programas y lograr que tengan efectos más perfeccionados. Este nivel de programación está, sin embargo, todavía muy lejos para quien no haya aprendido todavía su primer lenguaje de programación. Y aquí es donde entra este libro. Viene a enseñarle BASIC, el más popular de todos los lenguajes de microordenador.

Hay otra ventaja en la combinación de la electrónica y de la informática, pues no hay muchas aficiones, sean hobbys o vocaciones o una mezcla de ambas, en las que el microordenador pueda tener una utilidad tan clara. En electrónica se necesita hacer cálculos, aplicar fórmulas, generar datos de prueba y simular circuitos y, en todas estas áreas, el microordenador puede ser un valioso instrumento como quieren demostrar todos los programas presentados en este libro.

Para leer este libro es necesario disponer de un microordenador. Ahora es por lo tanto el momento de superar todos esos prejuicios contra la compra de equipos electrónicos ya montados y hacerse con un microordenador comercial. En principio, puede parecer menos atractivo que construírselo uno mismo, pero espero que, a la larga, encontrará abiertos nuevos y más gratificantes horizontes.

El libro nació de una serie de artículos de revista, titulados «Building BASIC», que aparecieron en la revista «Electronics and Computing Monthly». Tengo que dar las gracias a Dave Raven, su editor/director, por fundar una revista en la que la informática y la electrónica pueden convivir creativamente.

M.J.

1

LA TOMA DE CONTACTO CON EL MICROORDENADOR

La electrónica y los microordenadores están tan íntimamente relacionados que muchas veces resulta difícil decir dónde acaba una y dónde empieza el otro. Al fin y al cabo, un microordenador es una caja llena de dispositivos electrónicos y no hay en ella ningún componente oculto que el aficionado a la electrónica no haya encontrado en otros sitios.

Los microordenadores son una de las creaciones más interesantes de la electrónica moderna y por eso es lo más natural que cualquier persona interesada en la electrónica abarque también la informática. Frecuentemente pasa también lo contrario. Un interés que cubra sólo la utilización de los microordenadores es muy limitado. Se puede lograr que hagan cosas pero no se puede explicar cómo y porqué las hacen. El deseo de saber qué es lo que pasa en el interior de la caja puede llevar a algunos programadores a ahondar en el campo de la electrónica. Sin embargo, en esta era de la tecnología de la información la necesidad de saber utilizar un microordenador es más general que la necesidad de conocer cómo funciona y, ciertamente, es inconcebible que alguien que conozca el funcionamiento de un microordenador evite aprender a utilizarlo.

Podría parecer que el aficionado a la electrónica juega con ventaja cuando se pone a utilizar un microordenador porque está ya familiarizado con todas sus partes constituyentes. Por desgracia, no

es este el caso. El problema está en que un microordenador en funcionamiento no se parece en absoluto a nada electrónico. De hecho, los fabricantes han superado muchas dificultades para asegurarse de que los usuarios no puedan ver las interioridades de sus microordenadores. El empleo de un microordenador implica el desarrollo de unos conocimientos completamente nuevos: los conocimientos de programación.

No obstante, la combinación de la electrónica y de la programación tiene una ventaja definitiva. Si se sabe algo de electrónica, tanto analógica como digital, y después se adquieren conocimientos de programación, se estará en una mejor posición para utilizar los microordenadores con creatividad que si se es simplemente programador. Cuando se haya adquirido experiencia en ambos campos, se sabrá utilizar un microordenador y se conocerá el cómo y el porqué hace lo que hace.

El objeto de este libro es coger lo que cada uno sabe de electrónica y sumar a estos conocimientos los conocimientos de programación necesarios para aprovechar al máximo un microordenador. Para alcanzar este objetivo se irán describiendo muchas de las aplicaciones de la informática en el campo de la electrónica y, de paso, se aprenderá más electrónica. La facilidad que ofrece un microordenador de resolver cálculos complicados, sin ningún esfuerzo y con asombrosa precisión, es muy importante para el futuro de la electrónica. Y no sólo eso, sino que los ordenadores grandes pueden servir para diseñar circuitos integrados e incluso conjuntos electrónicos completos, desde los componentes y los circuitos impresos, hasta esquemas y paneles completos acabados. En última instancia, incluso diseñarán los robots que montarán todo lo demás. A un nivel más modesto, los microordenadores pueden efectuar cálculos de rutina como la selección de la resistencia limitadora de corriente para un diodo Zener. Estas aplicaciones menos espectaculares son, en cierta forma, más importantes. Utilizando un microordenador para hacer cálculos no sólo se puede ignorar la aritmética sino que, en cierta medida, se puede ignorar también todos esos símbolos innecesarios que quitan las ganas de aprender álgebra. Se puede decir que, al igual que la calculadora electrónica desterró la aritmética, el microordenador hará más accesibles las ideas que constituyen los fundamentos del

álgebra. La escritura de programas y la realización de cálculos con un microordenador puede ser una buena forma de sentirse más seguro en el mundo de la electrónica.

La necesidad de disponer de un microordenador

El mejor consejo que se le puede dar si no tiene la posibilidad de utilizar un microordenador es que compre uno. Y si ya tiene acceso a un microordenador, que no sea suyo, compre uno propio: recuperará la inversión con creces. Incluso en el caso de que no tenga una idea clara sobre todo esto del software, ahora es el momento de dar el paso decisivo porque la programación es una actividad y si lee este libro sin un microordenador al lado está haciendo el aprendizaje más difícil de lo que ya es. Todas las explicaciones y todos los programas-ejemplo incluidos son aplicables a uno de los equipos más baratos del mercado: el Sinclair ZX81. Aunque no es éste un «libro de programación del ZX81», no haría mal en salir y comprar este microordenador (después de todo, uno de los cursos BASIC tradicionales cuestan más que el ZX81).

Si había dado ya el paso adelante y ahora está preocupado por todo esto del ZX81 porque no es este el microordenador que ha comprado, tenga la seguridad de que éste también es su libro. Y si quiere comprar un microordenador con más prestaciones, como color y sonido, no se sienta cohibido al hacer su elección. Otros microordenadores son adecuados, aunque más caros, como el ZX Spectrum, el Commodore, el Dragon, el Oric, cualquiera de aquellos incondicionales Apple, o Tandy TRS-80, y cualquier otro microordenador CP/M con una versión BASIC Microsoft e incluso con los MSX.

La variedad de microordenadores es amplia porque el BASIC es, en la actualidad, un lenguaje casi normalizado y aceptado en versiones muy similares por casi todos los microordenadores que se puedan mencionar. Esto significa que la forma de BASIC utilizada en este libro será similar a la de los microordenadores más populares. Según se vaya entrando en materia, se irán indicando las singularidades del BASIC de los microordenadores más difundidos.

El principio

En la actualidad la mayoría de los microordenadores aceptan el lenguaje BASIC en una forma u otra. Pero el BASIC es solo uno de los muchos lenguajes evolucionados o de alto nivel que hay. Para saber qué quiere decir eso de «alto nivel» hay que partir de la idea de que el microprocesador hoy día es un dispositivo muy simple, que las acciones que puede realizar son pequeñas e insignificantes comparadas con el tipo de cosas que normalmente se quiere que haga. Por ejemplo, la (para nosotros) sencilla operación

1.5 × 2.5

exigirá que el microprocesador típico realice literalmente cientos de sus operaciones simples. El caso es que el microordenador sólo es capaz de hacer en una pasada verdaderas micro-cosas. Su poder procede no de una increíble capacidad o inteligencia sino de una increíble capacidad de operar muy, muy rápidamente. Para realizar una sola micro-acción puede tardar un microsegundo, lo que significa que se pueden conseguir un millón de micro-acciones en un segundo. Esa sencilla operación obligaría al microordenador a realizar miles de operaciones pero, aún así, sólo tardaría en hacerlas una milésima de segundo. En otras palabras, el microordenador es muy tonto pero muy rápido. Esto no es ninguna desventaja, ya que ¿qué sería mejor, escribir mil órdenes o escribir 1.5 × 2.5?

Los lenguajes de alto nivel

Ahí es donde entra en escena un lenguaje de alto nivel. Un lenguaje de alto nivel es una forma de definir (utilizando términos de la lengua inglesa y símbolos aritméticos) lo que se quiere que haga el microordenador, de manera que el propio microordenador pueda traducir este deseo en un gran conjunto de sus micro-acciones. La lista de micro-acciones resultante es lo que se suele conocer como «Código Máquina», que se dice que es un lenguaje de bajo nivel. Por lo tanto, se dan las instrucciones a la máquina en un lenguaje de nivel alto y se utiliza la máquina para traducirlas al lenguaje de bajo

nivel que entiende directamente. (Cuando se vean los compiladores y los intérpretes en el Capítulo 9 se dará más información sobre el proceso de traducción).

El BASIC es un lenguaje de alto nivel inventado mucho antes de que se soñara con los microordenadores. Era en principio un lenguaje para la enseñanza de la programación (BASIC viene de Beginners All-purpose Symbolic Instruction Code = Código de Instrucciones Simbólicas de Uso General para Principiantes). La idea era aprender a programar en BASIC y pasar después a alguno de los lenguajes más complejos como el FORTRAN o el ALGOL (en ese tiempo estos lenguajes se consideraban complejos). El problema fue que el BASIC era un lenguaje tan fácil de utilizar que la gente no quería pasar después a otros lenguajes. El resultado fue que se añadieron muchos «extras» a la forma inicial de BASIC para lograr un lenguaje mejor. La forma de estos «extras» variaba de un lugar a otro y por eso, en la actualidad, hay un núcleo de BASIC que es el mismo en todas partes y una serie de «extras» que pueden ser diferentes. Por lo tanto, el dialecto o tipo de BASIC que cada uno utiliza depende del microordenador que tenga pero, por lo menos, hay una coincidencia general en unas cuantas instrucciones básicas.

Con los microordenadores llegaron nuevas versiones del BASIC, la más conocida y extendida de las cuales fue descrita por una compañía de software norteamericana: Microsoft. La mayoría de los microordenadores más conocidos utilizan el BASIC Microsoft: los Pet, Oric, TRS-80, Video Genie, Apple, etc. Son equipos populares que no utilizan el BASIC Microsoft el Sinclair ZX81, el ZX Spectrum y el BBC Micro, pero incluso éstos utilizan formas de BASIC muy similares. Si este libro puede enseñar BASIC válido para tantos microordenadores distintos es por la popularidad del Microsoft.

La puesta en marcha del microordenador

Aunque el BASIC que utilizan la mayoría de los microordenadores es muy parecido, las formas en que lo utilizan son muy distintas. De ahora en adelante se partirá de la suposición de que no solo el microordenador que cada uno tiene pasa el BASIC sino de

que cada uno sabe cómo hacerlo. En algunos microordenadores esto es muy sencillo. En el caso del ZX81 y del ZX Spectrum, por ejemplo, lo único que hay que hacer es conectarlos. En otros, puede ser preciso cargar un programa en Código Máquina. Por ejemplo, algunos Apples se necesita que se cargue el BASIC por cinta y el Superbrain necesita que se cargue por disco el sistema operativo CP/M y el BASIC.

Lo que haya que hacer en cada caso se encontrará frecuentemente, tanto en el manual del microordenador, como aquí en la sección que lleva el título «puesta en marcha». Hablando de manuales, hay que decir que merece la pena tener a mano el manual de BASIC del microordenador mientras se está leyendo este libro porque, en lo relativo a los detalles concretos, es probable (pero no seguro) que sea más fiable que este libro. Tras leer cualquiera de los temas de los capítulos próximos de este libro es conveniente leer las secciones correspondientes del manual, tanto para comprobar que esa concreta versión BASIC funciona según se ha dicho, como —y esto es más importante— para acostumbrarse a leer el manual. (La lectura de los manuales de microordenadores es una habilidad muy parecida a hablar al revés. Muy pocos perseveran lo suficiente para adquirirla).

La introducción de instrucciones en el microordenador

Otra pequeña diferencia entre los distintos microordenadores es la forma de hacerles saber que se ha acabado de teclear una instrucción junto con las correcciones necesarias, etc. En muchos microordenadores lo único que hay que hacer es pulsar la tecla que lleva el nombre RETURN o CARRIAGE RETURN. Está claro que hay muchas formas de designar una tecla. Yo he visto las etiquetas ENTER, ACCEPT y NEWLINE (esta última en el ZX81, el ZX Spectrum y el Video Genie), todas ellas formas distintas de poner una etiqueta a la tecla que hay que pulsar para decir «he acabado de teclear y corregir la línea». Por tanto, al usuario corresponde descubrir cuál es la tecla de su microordenador. De ahora en adelante se supondrá que el lector la ha pulsado al acabar de teclear cada línea. Lo que el microordenador haga tras pulsar la tecla RETURN dependerá, por supuesto, de la línea que se haya pulsado pero, cuan-

do parezca que no pasa nada, hay que acordarse de la tecla «RETURN».

Teclados y errores

Además de la tecla especial RETURN la mayoría de los microordenadores tienen un conjunto de teclas que llenaría de vergüenza a una máquina de escribir. La distribución básica de todos los teclados es siempre la misma pero la ubicación de las teclas adicionales que llevan los microordenadores no ha sido nunca normalizada y, en consecuencia, en todos es diferente. Incluso un buen mecanógrafo es probable que tenga problemas para acostumbrarse a un teclado nuevo, y los que escriban con un dedo lo encontrarán incluso más difícil. Es importante no desalentarse en esta primera etapa porque aprender a utilizar un teclado es una de las habilidades más útiles que se pueden adquirir y es sorprendente la rapidez con que se aprende donde está cada tecla.

Un error que muchos principiantes cometen es confundir la dificultad de utilizar el teclado con la dificultad de usar el microordenador. Si se teclea mal una instrucción BASIC lo mejor que puede ocurrir es que no pase nada y, lo peor, que pase algo drástico que ciertamente no se quería. Todos los principiantes, e incluso algunos expertos cuando trabajan con un microordenador al que no están acostumbrados, pasan por la etapa de la torpeza de los dedos y es importante no confundir errores de mecanografiado con errores de programación. La diferencia es tan grande como la que hay entre ser capaz de sujetar un bolígrafo y hacer faltas de ortografía. No obstante, téngase la seguridad de que no hay ninguna posibilidad de averiar el microordenador por cometer cualquier tipo de error.

Los errores que puedan surgir cuando el lector teclee los ejemplos de este libro deben ser errores de mecanografiado. Luego, antes de pensar en lo peor, compruébese que el programa ha sido introducido correctamente. Si se ha revisado lo hecho y el programa sigue sin funcionar, la solución más probable es que la versión BASIC de ese microordenador sea diferente de la versión utilizada aquí. (Aunque no es probable, sí entra dentro de lo posible que se disponga de un microordenador que no sea tan conocido como los mencionados

anteriormente.) Compruébese en tal caso el manual de BASIC para averiguar dónde está la diferencia.

Aunque a lo largo de este libro se dan muchos ejemplos, la única forma de aprender BASIC es hacer prácticas y la única práctica adecuada es escribir programas. En las preguntas de autoevaluación del final de cada capítulo se encontrarán sugerencias para programas que cada uno puede escribir o mejorar. Es muy importante tomar en serio estos problemas. Ninguno de ellos es difícil (la verdad es que nada es difícil en BASIC) y hacerlos dará la seguridad de que se están siguiendo las ideas expuestas.

Ahora, a sacar el microordenador y a aprender BASIC...

2

LOS NUMEROS Y LA ARITMETICA

Es frecuente la idea de que la principal función de los microordenadores es hacer operaciones aritméticas cuando, en realidad, un microordenador medio emplea una proporción muy pequeña de su vida de trabajo haciendo sumas. Lo que sí hace se irá viendo más adelante. No obstante, descubrir la forma de trabajar cuidadosamente con números es un primer paso importante. Nos llevará, a su vez, a otros aspectos del BASIC.

Las operaciones y el orden

En BASIC están normalizadas las operaciones aritméticas usuales y el único cambio de la simbología convencional es el empleo de un signo «*» para la multiplicación. La razón de ello es que el signo «×» más normal puede confundirse con la letra «x» y los lenguajes de alto nivel, en ningún caso han de ser ambiguos. Además, el único signo de división que está permitido utilizar es el signo «/». Al fin y al cabo, sólo se necesita uno.

Una vez sentado lo de los cuatro operadores parecería que, por ejemplo, 4 + 3 ó 5 + 6/3*4.55 ó 6.7/5.8 + 5.6 ó —5/6*33.4 + 6.7 son todas piezas válidas de la aritmética BASIC. Pero todavía queda algo de ambigüedad. Hagamos una prueba realizando a mano una operación sencilla:

4 + 3*3

¿Cuál es la respuesta? ¿Veintiuno, que es cuatro más tres (o sea siete) tres veces (es decir, veintiuno)? ¿Trece, que es tres veces tres (o sea nueve) más cuatro (o sea, trece)? Si se hacen las sumas primero, siguiendo estrictamente el orden de izquierda a derecha, la primera respuesta es correcta. En cambio, si se hacen primero las multiplicaciones con independencia del sitio que ocupan en la escritura de la expresión, entonces trece es la respuesta.

En vez de discutir sobre cuál es la respuesta correcta es mejor pensar que a cada uno corresponde aceptar la forma de hacer las cosas que se propone. La aritmética no cayó del cielo. Es un invento del hombre y el hombre ha de decidir cómo se debe hacer. Y se da el caso de que hemos decidido efectuar las multiplicaciones primero. Por tanto, la segunda respuesta —13— es la correcta. Puede parecer más complicado de lo necesario, pero nosotros (es decir, los matemáticos) hemos decidido que la aritmética no se haga de izquierda a derecha siguiendo el orden estrictamente sino que el * y la / se deben hacer antes que la + y la —. Otra forma de decirlo es afirmar que el * y la / tienen «más prioridad» que la + y la — y que las operaciones de más prioridad se hacen antes. Cuando hay que optar entre signos de igual prioridad se hacen las operaciones de izquierda a derecha.

Sólo hay un detalle más que a estas alturas nos interesa: hay dos aplicaciones del signo menos. El signo menos normal es el que va entre dos números, por ejemplo 5 — 4 y, por lo tanto, se llama «operador binario». El otro significado del signo menos se da cuando va delante de un número, por ejemplo, —5. Este significa sencillamente «sustituir el número por su negativo». Estas dos aplicaciones del signo menos se prestan a confusión y por ello quizá fuera conveniente representar uno de ellos mediante otro signo pero ello induciría a pensar que la aritmética es muy distinta. Como los dos significados del signo menos se pueden distinguir muy fácilmente, en la práctica no hay problemas en seguir utilizando un signo para significar dos cosas. No obstante, hay que ir con cuidado y tener en cuenta que un número precedido del signo menos tiene más prioridad que los signos *, /, o + o un menos binario. Realícese por ejemplo, 5*—4. (La respuesta es —20).

Si esta idea de dar prioridades diferentes a los operadores aritméticos le parece difícil, siempre le queda la posibilidad de utilizar

paréntesis para hacer que el orden con el que quiere que sean hechas las operaciones quede perfectamente claro. Las operaciones aritméticas encerradas entre paréntesis se hacen siempre en primer lugar. Por ejemplo, si se quiere que la suma que hay en 3 + 2*2 se realice en primer lugar, póngase entre paréntesis: así, (3 + 2)*2 da 10. Los paréntesis pueden ser innecesarios en algunos casos, como en 3 + (2*2) pero nunca perjudican. Luego, ante la duda, utilice paréntesis.

Es posible que piense que este problema del orden de realización de las operaciones aritméticas es algo que nunca le dará preocupaciones. Espero que éste sea el caso pero, de todas formas, merece la pena señalar que uno de los errores más normales en ésto del BASIC es la incorrecta transferencia de las fórmulas del papel al microordenador. Por ejemplo,

$$\frac{1}{2 \times 3}$$

se suele pasar incorrectamente a BASIC con la forma 1/2*3. El motivo por el que esto es incorrecto debería ser obvio: la división se hace en primer lugar y después la multiplicación, dando (1/2)*3; pero hay que admitir que esa expresión se parece más a la que está descrita en el papel que la expresión en BASIC correcta, que es 1/2/3 (es decir, uno dividido por dos y luego dividido por tres). Si se quiere utilizar una forma que se parezca un poco más a la ecuación original no tiene más remedio que emplear el paréntesis, escribiéndola así: 1/(2*3).

En la aritmética que se requiere para los cálculos electrónicos se necesitan bastante más matemáticas que las cuatro operaciones aritméticas básicas. Pero no hay por qué preocuparse. El microordenador hace la mayor parte del trabajo y deja tiempo libre para pensar en lo que se está haciendo. Según se vaya avanzando, irá introduciéndose en el tema.

El primer programa en BASIC

Si tiene el microordenador preparado, teclee:

```
PRINT 2*2
```

(No hay que olvidarse de pulsar la tecla RETURN una vez escrita la línea). Si el microordenador funciona, aparecerá en la pantalla la solución: el 4. Lo que se acaba de poner al descubierto es que ese microordenador relativamente barato puede hacer las funciones de una calculadora cara. (Se puede teclear la operación aritmética que se quiera y el microordenador dará la solución. Hágase la prueba

```
PRINT 4 + 4*3
```

y se obtendrá la respuesta —16— en demostración de que el microordenador obedece las reglas de prioridad que se acaban de explicar.

También se ha puesto de manifiesto que ese microordenador puede aceptar un comando de BASIC, introducido por el teclado (PRINT es un comando BASIC) y actuar inmediatamente en consecuencia. Es el denominado funcionamiento en «modo inmediato». Probemos ahora con

```
10 PRINT 4 + 4*3
```

Esta vez no pasa nada al pulsar la tecla RETURN. El microordenador se queda a la espera de la introducción de otra línea por el teclado. Esto ocurre porque se ha comenzado la línea con un número —el número de línea— y esto hace que el comando sea almacenado en la memoria del microordenador pero sin producir ningún efecto. El microordenador trabaja en un «modo diferido». Para comprobar que la línea ha sido almacenada en memoria, hay que teclear

```
LIST
```

y la línea saldrá reproducida en pantalla. Para hacer que el microordenador ejecute la orden diferida, hay que teclear

```
RUN
```

y saldrá en pantalla el resultado: 16.

Un programa en BASIC es simplemente una lista de instrucciones diferidas. Cada instrucción tiene un número de línea distinto y cuan-

do el microordenador ejecuta el programa, al teclear RUN, va cogiendo las instrucciones, una por una, según el valor de su número de línea, empezando por las de los números más bajos. Luego, si se teclea

```
20 PRINT 2*2
```

seguido por

```
LIST
```

aparecerá en pantalla tanto la línea anterior 10 como la línea nueva 20. Si se teclea RUN saldrán en pantalla los números 16 y 4. Y éste es nuestro primer programa en BASIC.

Modos de funcionamiento del microordenador

En el apartado anterior se puso de manifiesto que el microordenador tiene dos modos distintos: el inmediato y el diferido. Esta es una idea importante que merece una explicación más detenida. Al conectar el microordenador, éste se pone a escrutar continuamente el teclado para descubrir lo que se haya tecleado. Según se van tecleando letras sueltas el microordenador las va juntando y guardando en un intento de componer la línea de BASIC que ha sido tecleada. En esta fase, el microordenador no está escrutando lo que está siendo tecleado. Se limita a actuar como un cuaderno de notas. Mientras se está tecleando el conjunto de letras se pueden hacer correcciones a voluntad, pero, una vez se ha pulsado la tecla RETURN, las cosas cambian: el microordenador examina la instrucción para descubrir qué es lo que se quiere que haga y ya no se pueden corregir los errores o cambiar de idea.

Lo primero que el microordenador busca es la presencia o ausencia de un número de línea. Si el comando carece de número de línea el microordenador la ejecutará inmediatamente, pero si el comando empieza con un número de línea, se interrumpe la operación. El microordenador se limita a ponerla junto a las anteriores instrucciones con número de línea ya almacenadas en memoria.

De esta forma, se puede componer una lista de instrucciones diferidas para uso posterior. Claro que esta lista de instrucciones diferidas no serviría para nada a no ser que hubiera alguna forma de decir al microordenador que las ejecute. Como se ha puesto de manifiesto anteriormente, la instrucción inmediata RUN hace que el microordenador comience a ejecutar la lista de instrucciones siguiendo el orden de sus números de línea en sentido creciente. Se podría decir que, mientras el microordenador ejecuta esta lista de instrucciones está en otro modo porque ya no se dedica a examinar el teclado para ver lo que está siendo tecleado. Se dedica a realizar el conjunto de sus propias tareas internas.

La edición de programas

Por lo tanto se puede componer un programa tecleando instrucciones en BASIC diferidas. Cada instrucción se puede editar antes de pulsar la tecla RETURN utilizando la tecla BACKSPACE (Retroceso) o DELETE (Borrado). Pero una vez pulsada la tecla RETURN, el microordenador almacena lo que se teclee. Esto será correcto siempre y cuando las instrucciones introducidas sean adecuadas. La forma más directa de modificar una instrucción, una vez introducida, es teclearlo todo otra vez. Si se teclea una línea con un número de línea que ya existe, la nueva línea sustituirá a la anterior. En otras palabras, tecleando 10 PRINT 3*3 se añadirá la nueva línea al programa incluso aunque esto signifique superponerla a una línea 10 existente.

Si se teclea una línea con un número de línea comprendido entre dos números de líneas ya existentes la nueva línea quedará insertada en el programa en el lugar correcto. Por tanto, si se tiene un programa compuesto por las líneas 10 y 20 y se quiere insertar una instrucción más entre ellas, basta utilizar el número de línea 15. Si se quiere borrar una línea basta teclear su número de línea y nada más. Es decir, al teclear 10 y pulsar la tecla RETURN se eliminará la línea 10 si es que estaba en el programa.

Si se quiere borrar todo el programa, se puede teclear el comando inmediato NEW. Pero mucho cuidado; en la mayoría de los microordenadores no hay forma de deshacer lo provocado por la

NEW. Tras realizar cada uno de los ejemplos de este libro, y antes de teclear el siguiente, no hay que olvidarse de borrar el programa anterior tecleando NEW. Si no se hace así, se mezclarán todos entre sí.

Hay formas más complicadas de editar programas pero varían de un microordenador a otro. Es conveniente, sin embargo, que cada uno descubra los comandos de edición de su microordenador. Algunos microordenadores tienen características que hacen la edición muy directa, mientras que otros son menos fáciles de utilizar. Aún así, una vez se ha logrado la familiarización con los mismos, la introducción de programas es muy fácil.

Las instrucciones del BASIC

La primera etapa del aprendizaje de un lenguaje de microordenador (como de cualquier otra lengua) es aprender unas cuantas palabras. Esto es lo que se hará aquí: presentar unas cuantas instrucciones sencillas y luego pasar a escribir un pequeño programa. En esta primera etapa sólo se tendrá en cuenta la forma más sencilla de cada instrucción, lo que significa que según se vaya avanzando y se intente escribir programas más complicados, habrá que ver de nuevo las distintas instrucciones.

La instrucción PRINT

He aquí la primera instrucción de BASIC. La forma de la instrucción PRINT es

PRINT lista a imprimir

donde «lista a imprimir» es simplemente lo que se quiere que aparezca en pantalla. El tipo de cosas que se puede incluir en una lista a imprimir es muy amplio. Según se vaya avanzando se irán viendo nuevos tipos de cosas. Las más importantes sin embargo, son las «cadenas literales» y las «variables».

Las cadenas y las variables

Una cadena literal es lo que se introduce por el teclado incluido entre comillas. Los caracteres entre comillas, es decir la cadena, salen impresos literalmente en la pantalla.

Una variable es el nombre de una zona de la memoria del microordenador en la que el usuario puede almacenar datos. Algunos microordenadores permiten el empleo de nombres largos pero sólo teniendo en cuenta las dos primeras letras del nombre. Otros sólo permiten nombres de dos letras, algunos solo de una letra y un número (por ejemplo la F3) y los más sencillos sólo permiten nombres de una letra. Para simplificar las cosas, aquí se usarán sólo nombres de una letra, pero si un microordenador puede operar con nombres de más de una letra tanto mejor, pues esto significa que es posible utilizar nombres de variables que identifiquen por sí mismos la naturaleza de lo que está almacenado en cada variable. Las variables sirven para almacenar números para usar en posteriores cálculos. La forma exacta de almacenar números en ellas se explicará más adelante.

Los diferentes elementos de una instrucción PRINT han de estar separados entre sí pues, en caso contrario, no se sabría donde empieza uno y acaba otro, y además el BASIC da especial significado a los medios de separación empleados. Si se utiliza una coma deja un cierto número de espacios en blanco entre cada elemento y si se emplea un punto y coma no deja espacios en blanco.

Ha llegado el momento de probar algunas instrucciones PRINT. Tecléese:

```
10 PRINT "HOLA ESTO ES UNA CADENA LITERAL"
20 PRINT "UNA", "DOS"
30 PRINT "UNA"; "DOS"
40 PRINT "A",A
RUN
```

Cada instrucción PRINT se refiere a algún punto concreto; la línea 10 se limita a imprimir la cadena literal; las líneas 20 y 30 muestran la forma de usar la coma y el punto y coma para determinar la separación de espacios en blanco; la línea 20 imprime algo así:

UNA DOS

pero la línea 30 imprime

UNADOS

La línea 40 muestra la diferencia entre una literal y una variable. La A entre comillas aparece en la pantalla pero la segunda A no está encerrada entre comillas y es el nombre de una variable. En algunos microordenadores esta nueva línea provocará un mensaje de error porque se ha intentado imprimir el contenido de un área de memoria llamada A cuando en ninguna parte de este programa se ha almacenado nada en ella. ¿Se puede esperar que el microordenador imprima el contenido de una variable que nunca se ha utilizado? No obstante, en muchos microordenadores la línea 40 hará que salga impreso:

A 0

¿De dónde viene el cero? A es una variable y PRINT A significa imprimir el contenido de la parte de memoria del microordenador llamada A. Lo que sale impreso por lo tanto depende de lo que hay en A. El BASIC, por convención, pone a cero todas las variables al teclear RUN. Es importante tener en cuenta que esta puesta a cero automática es algo que no está ni mucho menos normalizado en el BASIC. Por eso es una buena costumbre no dar por sentado que una variable tiene un valor inicial cero sino dedicarla explícitamente a algo. Si se sigue esta regla de definir siempre el contenido de todas las variables utilizadas en un programa, se verá que hay más probabilidades de que los programas hechos por uno mismo pasen en otra versión de BASIC.

La instrucción INPUT

INPUT es, en cierto sentido, lo contrario de PRINT. Pide al usuario que suministre información y coloca esta información en una variable para su uso posterior por el programa. Por ejemplo,

```
10 INPUT A
20 PRINT A
RUN
```

debe provocar la aparición de un signo de interrogación en la pantalla. Si se teclea un número, por ejemplo el 123, y se pulsa la tecla RETURN, el número debe salir impreso en pantalla en la línea siguiente. INPUT A significa «coger un número del teclado y guardarlo en la zona de memoria de nombre A», es decir, en la variable A, y ya se sabe lo que hace PRINT A. Adviértase que INPUT hace que el microordenador deje de ejecutar el programa y empiece a examinar el teclado. Seguirá examinando el teclado hasta que alguien teclee un número y pulse la tecla RETURN. Entonces almacenará el número en la variable definida en la instrucción INPUT y seguirá con el resto del programa.

En la mayoría de los microordenadores (pero no en el ZX81) se puede emplear una forma más avanzada de instrucción INPUT. La instrucción INPUT puede pedir la introducción de más de una variable cada vez. La forma general es

```
INPUT lista de entrada
```

La lista de entrada es simplemente una lista de variables separadas por comas. Tecléese ahora:

```
10 INPUT A,Z
20 PRINT A,Z
RUN
```

Cuando aparece el signo de interrogación en pantalla el microordenador está pidiendo dos números. Se podría responder tecleando, por ejemplo,

```
2.3,4.5
```

lo que provocaría la aparición en la línea siguiente de la pantalla de

```
2.3          4.5
```

Obsérvese que los dos números se introdujeron juntos pero separados por comas. Algunos microordenadores permiten que el usuario teclee un número y luego piden el siguiente con el signo de interrogación. Veamos qué hace nuestro microordenador si se vuelve a pasar el programa y se introducen tres o cuatro números a la vez.

La instrucción LET

La instrucción INPUT es una forma de poner información en las variables. La instrucción de asignación es otra. Su forma general es

LET variable = expresión

donde variable puede ser cualquier variable válida y expresión puede ser cualquier expresión aritmética válida. Tecléese:

```
10 LET A = 4.5
20 PRINT A
RUN
```

En la pantalla debe aparecer el número 4.5.
Pruébese ahora con

```
10 LET B = 3.14159
20 LET R = 2
30 LET A = B*R*R
40 PRINT "EL AREA DE UN CIRCULO DE RADIO ";R;"
     ES ";A
RUN
```

Si se recuerda que el área del círculo es pi (3.14159) veces el radio al cuadrado y que elevar al cuadrado es lo mismo que multiplicar un número por sí mismo, no debe haber problemas con este ejemplo. La única característica nueva es el empleo de variables dentro de expresiones aritméticas. Este punto se analiza en el apartado siguiente.

Merece la pena mencionar que la mayoría de las versiones de BASIC (pero no el BASIC ZX) permiten no poner LET, con lo que se ahorra mucho trabajo con el teclado.

Las variables

La idea de variable ha sido presentada aquí como parte de la explicación de las instrucciones del BASIC INPUT, PRINT y LET. Aunque la idea de variable no es una idea complicada puede provocar confusiones hasta que uno se acostumbra a ella. Una variable, es una zona de la memoria del microordenador que tiene un nombre y que sirve para almacenar datos. Los datos almacenados en una variable existen solo mientras el programa está en marcha, y se pierden, casi con total seguridad, si se desconecta el microordenador. Es posible volver a usar variables almacenando en ellas datos nuevos pero quedarán superpuestas sobre el contenido anterior, borrado. Por ejemplo:

```
10 A = 1
20 PRINT A
30 A = 2
40 PRINT A
```

almacena el 1 en A, lo imprime, almacena luego el 2 en A y lo imprime. Una forma muy gráfica de ver una variable es considerarla como una caja con una etiqueta en la que se puede almacenar un número.

Tal como se ha visto en el apartado anterior, cabe la posibilidad de almacenar un valor en una variable utilizando la instrucción de asignación o la instrucción INPUT, y se puede averiguar qué hay almacenado en una variable utilizando la instrucción PRINT. No obstante, con una variable se pueden hacer más cosas que simplemente almacenar y recuperar valores. Se puede utilizar una variable en una expresión aritmética exactamente igual que si fuera un número. Así se hizo en el área del círculo del apartado anterior, cuando se tecleó

```
30 LET A = B*R*R
```

y dio el resultado de multiplicar los contenidos de R y B entre sí.

En otras palabras, las expresiones del tipo X*Y se realizan con los contenidos actuales de las variables X e Y. Cabe también la

posibilidad de mezclar variables y números componiendo expresiones como 3*A/B + 20. De hecho, cabe la posibilidad de emplear una variable donde esté permitido utilizar un número.

Todo esto parece muy fácil, pero hay que tener una mente muy lúcida para descifrar el significado de algo como:

```
10 A = 1
20 PRINT A
30 A = A + 1
40 PRINT A
```

Si se observa la línea 30 se ve que la A aparece a ambos lados del signo igual. La línea 10 almacena un 1 en A y la línea 20 lo imprime. ¿Qué hará la línea 30? Es decir, ¿qué imprimirá la línea 40? La línea 30 simplemente suma un 1 al número almacenado en ese momento en A. En el momento de hacer el cálculo del término que hay a la derecha del signo igual, A contiene un 1 y, por lo tanto, el resultado es un 2. Una vez obtenido este resultado, es almacenado en A, superponiéndolo, por lo tanto, al valor anterior, y el valor impreso por la línea 40 es lógicamente un 2. Una variable puede aparecer en el lado derecho y en el izquierdo de una instrucción de asignación, en cuyo caso hay que pensar que en primer lugar es tratado el término de la derecha y después es almacenado el resultado en la variable del lado izquierdo.

Cálculo de los valores de resistencias

A título de ejemplo de lo que se puede hacer con las tres instrucciones vistas hasta ahora, intentemos calcular la resistencia total de dos resistencias en serie. Si tenemos dos resistencias de A ohmios y B ohmios, la resistencia total si están conectadas en serie es simplemente A + B ohmios. Tecléese

```
10 PRINT "CUAL ES EL VALOR (EN OHMIOS) DE LA
       PRIMERA RESISTENCIA"
20 INPUT A
30 PRINT "CUAL ES EL VALOR DE LA SEGUNDA
       RESISTENCIA"
```

```
40 INPUT B
50 LET C = A + B
60 PRINT "RESISTENCIA SERIE TOTAL = ",C
RUN
```

Fácil ¿no?

En el caso de resistencias en paralelo, la resistencia total es el resultado de una expresión más compleja (A*B/(A + B)) pero precisamente por eso merece la pena escribir el programa que lo haga. Modifíquese el programa anterior tecleando las líneas siguientes:

```
70 LET C = A*B/(A + B)
80 PRINT "RESISTENCIA TOTAL EN PARALELO = ",C
```

Está claro que se puede modificar el programa para operar con condensadores pero resulta ya más difícil ampliarlo para que abarque el caso de más de dos resistencias. Se necesita algo más de BASIC para hacerlo.

Problemas con potencias

En el programa del área del círculo presentado anteriormente surgía la necesidad de calcular el cuadrado de un número. La forma de hacerlo consistió en volver a la definición de cuadrado de un número como X*X. Pero a veces hay que calcular una potencia que no se puede poner en la forma de una multiplicación repetida. Por esta razón la mayoría de las versiones de BASIC incluyen un símbolo «elevar a la potencia». Pero en la actualidad se emplean dos símbolos: el doble asterisco (**) y la flecha hacia arriba (↑). Averigüe cuál utiliza la versión de su BASIC (en este libro se empleará la flecha de aquí en adelante) y haga la prueba siguiente. Teclee:

```
10 INPUT A
20 INPUT B
30 C = A ↑ B
40 PRINT C
```

(No hay que olvidarse de utilizar el símbolo correcto en la línea 30). Emplee este programa para averiguar con qué valores de A y B opera su versión de BASIC. Muchas veces el campo de valores posibles no es tan amplio como era de esperar debido a las limitaciones implícitas en la forma de calcular las potencias. Cuando va junto a otras operaciones aritméticas, la elevación a una potencia (llamada también exponenciación) tiene la mayor prioridad y, por lo tanto, es calculada siempre en primer lugar. Por ejemplo, 3 + 2↑2 es 3 + (2↑2), es decir, 7, y 3*2↑2 es 3*(2↑2), es decir 12. Cabe la posibilidad de elevar números a la potencia dada por una expresión, por ejemplo, 2↑(1/2) sacará la raíz cuadrada de 2.

Es importante pensar que, aunque el microordenador hace las operaciones aritméticas a la velocidad de la luz, las potencias son difíciles de calcular (implican calcular el logaritmo, hacer una multiplicación y calcular el antilogaritmo) por lo que conviene evitarlas siempre que sea posible. Luego, en vez de A↑2 es mejor poner A*A, en vez de A↑3 es mejor poner A*A*A, y así sucesivamente. No obstante, todo tiene su límite, y escribir A↑50 es ciertamente mejor que escribir A cincuenta veces.

Preguntas de autoevaluación

1. Resuélvanse las siguientes expresiones aritméticas (utilizando una calculadora pero no un microordenador).

(a) 2*3 + 11
(b) 3 — 4/8
(c) 6*35/25
(d) 1/2 + 2
(e) 1/(2 + 2)

2. Escríbase un programa corto para introducir la longitud de un lado de un cuadrado y sacar impresa su área.

3. Modifíquese el programa del ejercicio **2** de manera que calcule el área de un rectángulo.

4. El número de espiras de un solenoide de núcleo de aire de inductancia L μH viene dado por

$$\frac{0.4Ld + 0.16L^2d^2 + 0.0033r^3L}{0.0031r^2}$$

donde d es el diámetro del hilo en milímetros y r es el radio del núcleo en milímetros. Escríbase un programa que imprima el número de espiras de una inductancia concreta dado el diámetro del hilo y el radio del núcleo. Utilícese este programa para investigar las posibilidades de emplear una bobina de núcleo de aire en un circuito sintonizado que necesita una inductancia de 10 μH.

3

LAS INSTRUCCIONES DE CONTROL

Al final del Capítulo 2 se había aprendido suficiente BASIC como para poder escribir programas. No obstante, aunque se trataba de programas de innegable utilidad, eran apenas diferentes del tipo de soluciones que pueden obtenerse sin muchos problemas con una calculadora. En este Capítulo se va a tratar de la verdadera esencia de la programación de microordenadores: las instrucciones que alteran el orden de ejecución de otras instrucciones.

El flujo de control

Los programas escritos hasta ahora eran ejecutados siguiendo el orden de sus números de líneas empezando por la instrucción con el número de línea más bajo. Esto corresponde a ejecutar el programa leyéndolo línea a línea de arriba abajo. El orden de ejecución de las instrucciones recibe el nombre de «flujo de control», porque, mientras es ejecutada cada instrucción, esa instrucción está en posesión del «control» de lo que está pasando, y el control «fluye» de línea a línea recorriendo el programa del principio al final.

Aunque este sencillo flujo de control es fundamental para la programación, es inadecuado para la mayoría de los problemas. Considérese, por ejemplo, la tarea de sacar impresa la serie de números 1,2,3... El único método conocido a estas alturas del libro da algo así:

```
10 A = 1
20 PRINT A
30 A = A + 1
40 PRINT A
50 A = A + 1
60 PRINT A
...
...
```

Empieza asignando a A el valor 1 y lo imprime en la línea 20, después suma una unidad para hacer A 2 y en la línea 40 lo saca impreso, y así sucesivamente. Está claro que el programa es tan largo como aburrido. Lo que se necesita para resolver este problema tan sencillo es repetir de alguna forma la misma acción: sumar una unidad a A e imprimir A, una y otra vez. Se necesita un bucle.

El bucle y GOTO

La instrucción GOTO es una forma de hacer que el microordenador ejecute una instrucción que puede no ser la siguiente de la secuencia. La forma de la instrucción GOTO es simplemente:

GOTO número de línea

o

GO TO Número de línea

en función del microordenador que se posea y de la versión que se prefiera. La mayoría de los microordenadores aceptan las dos (el ZX81 lleva escrita en una de sus teclas la expresión GOTO pero presenta en pantalla GO TO). El efecto de GOTO es fácil de ver en el pequeño programa siguiente:

```
10 PRINT "ESTA ES LA LINEA 10"
20 GOTO 40
30 PRINT "ESTA ES LA LINEA 30"
```

```
40 PRINT "ESTA ES LA LINEA 40"
50 PRINT "ESTA ES LA LINEA 50"
```

El resultado de ejecutar este programa es obviamente la ejecución de las líneas 10, 40 y 50. Esto muestra que el microordenador salta la línea 30 al ejecutar la GOTO 40 y que, tras la línea 40, el flujo de control vuelve a la normalidad: no vuelve a la línea 20 sino que pasa a la línea siguiente, la 50. Es decir, el microordenador no se acuerda de haber ejecutado una instrucción GOTO, y, a no ser que esté ejecutando una instrucción GOTO, el control fluye de línea a línea hacia el final del programa. ¿En qué forma nos sirve GOTO para repetir operaciones? Tecléese:

```
10 A = 1
20 PRINT A
30 A = A + 1
40 GOTO 20
RUN
```

La pantalla del microordenador se llenará de números, uno por línea, porque la instrucción GOTO de la línea 40 transfiere el control a la línea 20 y el programa estará dando vueltas continuamente. El problema ahora es parar; algunos microordenadores se paran sacando un mensaje de error cuando el número llega a ser demasiado grande o la pantalla está llena y, en otros, hay que pulsar las teclas STOP o RESET. Pero ninguna de las dos es una buena forma de parar. Si se quiere ver en la pantalla una cantidad incluso mayor de números, hay que hacer la prueba poniendo

```
20 PRINT A;
```

Si se quiere ver una secuencia de números diferente hay que poner

```
30 A = A + A
```

o

```
10 A = 2
30 A = A*A
```

Esta repetición de una acción sencilla es uno de los métodos de programación fundamentales; su nombre técnico es «iteración».

Las expresiones condicionales

Algunas veces se desea que el flujo de control cambie en función de alguna condición. Por ejemplo, una forma de parar limpiamente el anterior programa de conteo consiste en ejecutar la instrucción GOTO solo si A es menor que 10, por ejemplo. Así sacaría impresos los números del 1 al 9 y después se pararía. Esto se puede hacer mediante una instrucción IF. Tecleando

```
10 A = 1
20 PRINT A
30 A = A + 1
40 IF A<10 THEN GOTO 20
RUN
```

deberían salir en pantalla los números del 1 al 9. La línea 40 es una instrucción IF típica. Entre IF(SI) y THEN (ENTONCES) se puede escribir una amplia variedad de condiciones. Por ahora, nos limitaremos a las más sencillas. Si la condición es cierta (si se cumple), es ejecutada la instrucción que sigue a THEN —en este ejemplo la instrucción GOTO 20—. Si la condición no es cierta, el flujo de control pasa a la línea siguiente como siempre. En este ejemplo el programa acabaría por ausencia de más instrucciones. Por lo tanto, la forma general de la instrucción IF es:

IF condición que puede ser cierta o falsa THEN alguna instrucción

Hay que destacar que la instrucción que sigue a THEN puede ser cualquier instrucción de BASIC y no solo GOTO como en el ejemplo anterior. De hecho, la instrucción que sigue a THEN puede incluso ser otra instrucción IF en algunas formas de BASIC.

La condición puesta entre IF y THEN es conocida como expresión «lógica» o «condicional». En BASIC simple toma la forma:

expresión aritmética operador relacional expresión aritmética

Por ejemplo, A + 3 = Z es una expresión condicional. Las expresiones aritméticas son A + 3 y Z y el operador relacional es =. Los operadores relacionales normalizados en BASIC son:

= igual
< menor que
> mayor que
<= menor o igual que
>= mayor o igual que
<> no igual que

Los siguientes ejemplos de instrucciones IF muestran algunas de las formas en que pueden utilizarse:

```
IF A = Z + 3 THEN PRINT "Z ES TRES UNIDADES
        MAYOR QUE A"
IF Z<10 THEN Z = A + Z
IF Z<>0 THEN GOTO 50
```

He aquí una primera introducción a la instrucción IF. En el Capítulo 9 se verán sus formas más elaboradas.

La instrucción de paro de programa

Una vez vista la instrucción IF sería útil poder escribir «IF alguna condición THEN realícese el paro del programa». Lo que se necesita es una instrucción STOP. La única complicación es que unos BASICs usan END, otros emplean STOP, y algunos ambos. Averigüe, por lo tanto, cuál reconoce su microordenador con el significado «paro de programa». En el resto del libro se usará STOP.

Los bucles FOR

Tenemos ya casi todas las instrucciones de BASIC necesarias para escribir casi cualquier programa. No obstante, hay algunos tipos de bucles que aparecen con tanta frecuencia que merece la pena tener una instrucción simple para hacerlos. Concretamente, se pre-

senta con frecuencia la necesidad de ejecutar un cierto número de veces la misma sección del programa. Por ejemplo, para imprimir la cuarta potencia de dos (2*2*2*2) se podría utilizar la secuencia

```
10 N = 4
20 A = 2
30 I = 1
40 A = A*2
50 I = I + 1
60 IF I< =3 THEN GOTO 40
70 PRINT "LA";N;" POTENCIA DE DOS ES";A
```

La línea 10 pone N a la potencia de dos buscada y la línea 20 inicializa A a 2. I es una variable especial nueva que va a ser utilizada para contar el número de veces que se ejecuta el bucle. Tal variable de recuento recibe con frecuencia el nombre de variable índice. La línea 40 efectúa la operación 2*2 y, como es la primera pasada por el bucle, I es 1. La línea 60 comprueba si ha sido realizado el número suficiente de pasadas por el bucle y devuelve el control a la línea 40 si se han hecho menos de tres pasadas. Hay que destacar que en la línea 60 el índice I es una unidad más que el número de pasadas por el bucle. El resultado es que se hace la instrucción 40 tres veces, dando 2*2 = 4 en la primera pasada, 4*2 = 8 en la segunda pasada y 8*2 = 16 en la tercera.

Lo que se quería hacer era ejecutar la línea 40 para I = 1,2,3. A este efecto concreto el BASIC tiene el bucle FOR:

—FOR variable índice = expresión uno TO expresión dos
...
cualquier lista de instrucciones BASIC
...
NEXT variable índice

Las dos instrucciones FOR y NEXT juntas forman un bucle FOR. Al topar con una instrucción FOR el microordenador realiza las dos expresiones aritméticas y pone la variable índice al valor de la primera expresión y guarda el de la segunda para uso posterior (en un área de memoria llamada la pila FOR). La instrucción NEXT

simplemente marca el final del bucle. Al topar con una instrucción NEXT el microordenador incrementa la variable índice en una unidad y, si todavía es menor o igual al valor de la expresión dos, transfiere el control a la instrucción que sigue a la FOR.

Con un bucle FOR, el programa de las potencias de dos se podría hacer así:

```
10 N = 4
20 A = 2
30 FOR I = 1 TO 3
40 A = A*2
50 NEXT I
70 PRINT "LA";N;"POTENCIA DE DOS ES";A
```

que por supuesto da el mismo resultado que el ejemplo anterior.

Como las expresiones de la instrucción FOR pueden ser cualquier expresión aritmética válida, se puede utilizar la línea:

```
30 FOR I = 1 TO N —1
```

Con la ventaja adicional de que ahora se puede sustituir la línea 10 por

```
10 INPUT N
```

y se tiene un programa que saca la potencia enésima de dos.

A título experimental averigüe el valor máximo que puede tomar N en su microordenador.

La variable índice de un bucle FOR puede hacer también las funciones de una variable de BASIC normal sujeta a una regla: una variable índice no puede ser modificada durante un bucle FOR. Luego:

```
10 FOR I = 1 TO 10
20 PRINT I
30 NEXT I
```

es correcto, pero

```
10 FOR I = 1 TO 10
20 I = 3
30 NEXT I
```

no lo es.

Hay unas cuantas restricciones más que afectan a los bucles FOR. Algunas dependen del microordenador con el que se esté trabajando pero dos casi universales son:

(*a*) no se puede salir de un bucle FOR activo; y
(*b*) no se puede entrar a un bucle FOR y encontrar a continuación una NEXT.

La primera restricción significa que, si durante la ejecución de un bucle FOR, se descubre (por medio de una IF) que ya está hecho lo que se quería hacer, realizar otra pasada por el bucle FOR sería una pérdida de tiempo. Por lo tanto, hay que seguir adelante. (La causa de esto es que, solo cuando se ha acabado adecuadamente un bucle FOR, es eliminada la segunda expresión de la pila FOR). La segunda restricción es también necesaria porque si se topa con una instrucción NEXT como resultado de una entrada a un bucle FOR sin haber ejecutado la instrucción FOR, el microordenador no sabe qué hacer a continuación. Cabe la posibilidad de entrar y salir de un bucle FOR sin encontrar ninguna instrucción NEXT y no cometer, por lo tanto ningún error, pero esta forma de programación no se considera elegante. Siempre hay una forma mejor de resolver la situación.

Bucles FOR avanzados

La forma del bucle FOR presentada anteriormente sirve para la mayoría de las aplicaciones, pero hay una forma del bucle FOR que algunas veces puede ser de suma utilidad. En un bucle FOR normal, el valor de la variable índice es incrementado en una unidad cada vez. Pues bien, poniendo la instrucción STEP se puede hacer que la variable índice sea incrementada en la cantidad que se quiera. La forma general es:

FOR índice = expresión uno TO expresión dos STEP expresión tres

...

cualquier lista de instrucciones de BASIC

...

NEXT variable índice

Hacer la prueba con:

```
10 FOR I = 1 TO 10 STEP .1
20 PRINT I
30 NEXT I
```

que es un ejemplo de paso (STEP) o incremento de valor 0.1. Es decir, cada pasada por el bucle produce el incremento de la variable índice en 0.1, hasta que pasa a ser mayor que 10. Podría surgir la pregunta: ¿cuál es el último valor de I que se imprime? No hay ningún truco en el empleo de pasos (STEP) fraccionarios excepto recordar que el bucle FOR acaba en el mismo momento en que el valor índice es mayor que el de la segunda expresión. No obstante, es posible especificar un paso negativo. Por ejemplo

```
10 FOR I = 10 TO STEP —1
20 PRINT I
30 NEXT I
```

En este caso, en cada pasada por el bucle se resta una unidad de la variable índice. El único cambio es que ahora el bucle acaba cuando la variable índice es menor que la segunda expresión.

Averiguar cuando acaban los bucles FOR es muy fácil siempre y cuando se tengan en cuenta las dos reglas: (*a*) si el paso es positivo el bucle acaba cuando el índice es mayor que la segunda expresión; y (*b*) si el paso es negativo el bucle acaba cuando el índice es menor que la segunda expresión.

Los bucles FOR pueden empezar a parecer un poco complicados cuando se utilizan valores negativos en sitios distintos a los pasos pero, teniendo en cuenta que cuanto más negativo es un número menor es (por ejemplo, que —2 es mayor que —3) la aplicación de las dos reglas no presenta ningún problema. Por ejemplo,

```
10 FOR I = —100 TO —10 STEP + 1
20 PRINT I
30 NEXT I
```

imprimira los números —100, —99 hasta —10 (porque —9 es mayor que —10). Y sustituyendo la línea 10 por

```
10 FOR I = —10 TO —100 STEP —1
```

se imprimirán los números comprendidos entre —10 y —100 (porque —101 es menor que —100).

El caso en el que se salta por encima del bucle FOR

Hay un problema asociado al empleo de los bucles FOR que es consecuencia de que no haya un BASIC estándar. Considérese (y hágase en el microordenador a efectos de prueba) el siguiente programa:

```
10 INPUT C
20 INPUT F
30 FOR I = C TO F
40 PRINT I
50 NEXT I
60 GOTO 10
```

Se trata de un bucle FOR muy simple que opera con valores de comienzo y final almacenados en las variables C y F. Si se introducen valores como 1 para C y 10 para F se verá impreso el conjunto de números que era de esperar. Pero ¿qué pasa si se hace la prueba con un valor de 1 para C y un valor de 1 para F? La respuesta no es, en absoluto inesperada, si se han aplicado bien las reglas que rigen el funcionamiento de los bucles FOR. El bucle es ejecutado una vez, y se imprime un 1 por efecto de la línea 40 porque, cuando el programa llega a la instrucción NEXT, suma una unidad a la variable índice poniéndola a 2, mayor por lo tanto que el valor de F.

Considérese ahora el mismo problema pero con un valor de 2 para C y de 1 para F. Esta vez las cosas no son tan claras porque la variable índice comienza con un valor que es superior a F (porque 2 es mayor que 1). ¿Podría ejecutarse el bucle una vez y producirse el paro del mismo, después de haberse impreso el valor 2 en la línea 40? (Téngase en cuenta que la instrucción NEXT suma una unidad a la variable índice poniéndola a 3 que es, sin duda, mayor que 1). La respuesta no está clara y esta falta de claridad queda reflejada por la diversidad de resultados que dan las diferentes versiones de BASIC (unas ejecutan el bucle una vez y otras ninguna). En la práctica lo más frecuente es que se salte por encima del bucle FOR cuando el valor inicial es mayor que el valor final. Cuando se escribe un programa en el que se dé este caso, recúrrase a esta solución:

```
10 INPUT C
20 INPUT F
30 IF C>F THEN GOTO 70
40 FOR I = C TO F
50 PRINT I
60 NEXT I
70 STOP
```

se tendrá la seguridad de que el programa funciona con independencia de lo que hiciera la versión BASIC en cuestión. Es conveniente prestar siempre una atención especial a los bucles FOR que tienen valor inicial y valor final para asegurarse que todo se desarrollará bien.

Los bucles anidados

Cuando se empiezan a escribir programas más complicados se quiere poner más de un bucle FOR por programa. Las reglas que rigen el empleo de múltiples bucles FOR pueden parecer difíciles, pero basta recurrir a los anidamientos. Si se tienen dos o más bucles FOR en un programa y la ejecución de uno termina antes

del comienzo de la ejecución del siguiente, entonces no puede haber problemas. He aquí el modelo:

FOR I...NEXT I...FOR J...NEXT J...

Es siempre correcta. Pero si se comienza un bucle FOR *dentro* de otro, las cosas pueden ir mal. La regla a seguir es que los bucles FOR no se pueden cruzar: deben estar totalmente anidados unos en otros.

Por ejemplo, la forma FOR I...FOR J...NEXT I...NEXT J no es aceptable porque el primer bucle acaba antes que el segundo y, por lo tanto, el segundo no está contenido en el primero. La forma FOR I... FOR J... NEXT J... NEXT I es correcta. No habrá errores, siempre y cuando los bucles FOR internos estén completamente contenidos en los bucles FOR externos, partiendo de que el microordenador en cuestión pueda operar con tal número de bucles FOR (el ZX81, por ejemplo, puede operar hasta con 26). Los bucles FOR anidados solo tendrán razón de ser cuando se aprenda algo sobre matrices en el capítulo siguiente. Si, de todas formas, el lector está impaciente por conocer los efectos de una pareja de bucles FOR anidados, haga la prueba con:

```
10 FOR I = 1 TO 10
20 FOR J = 1 TO 10
30 PRINT I,J
40 NEXT J
50 NEXT I
```

Sirve de ayuda para comprender qué es lo que está pasando, pensar que el bucle interior (el que utiliza la J como variable índice) empieza en 1 y sigue hasta 10 cada vez que se hace una pasada por el bucle exterior (el que utiliza la I como variable índice).

Empleo de las instrucciones GOTO, IF y FOR

Las tres instrucciones de BASIC presentadas en este capítulo son muy fáciles de entender pero, por desgracia, entenderlas no es sufi-

ciente para escribir programas en BASIC. Hay que aprender cómo y cuándo utilizarlas, algo que normalmente solo viene con la práctica. Hay, sin embargo, algunos puntos generales que pueden acelerar este proceso y evitar con un poco de suerte la mala utilización de las instrucciones.

Hay que utilizar la instrucción GOTO para formar bucles allá donde sea necesario. No hay que emplearla para dar saltos repetidos y sin sentido por el programa. No suele haber necesidad de ello. La instrucción IF ha de utilizarse para salir de un bucle cuando se cumple una determinada condición, para pasar por alto una sección del programa o para elegir entre dos partes alternativas del programa. El bucle FOR ha de emplearse siempre que haya que ejecutar una sección de un programa un determinado número de veces. Si hay que ejecutar un bucle hasta que se cumpla una determinada condición, hay que emplear la instrucción GOTO para formar el bucle y la instrucción IF para salir de él; nunca para salir de un bucle FOR.

Empleo del temporizador 555

Como ejemplo de un programa real con algunas de las instrucciones presentadas en este capítulo, veremos una aplicación al diseño de temporizadores a base del circuito integrado 555. La figura 3.1 muestra un circuito temporizador típico del 555 para los que no estén familiarizados con este dispositivo. Un impulso en la línea de disparo (es decir, el cierre del conmutador) produce una salida alta durante un tiempo dado por la expresión

$$T = 1.1 \times R \times C \text{ ms}$$

donde R está en kilohmios y C en microfaradios.

El diseño de un temporizador es una tarea muy sencilla. Lo único que hay que hacer es averiguar los valores de R y C que dan el tiempo buscado. No obstante, hay que satisfacer las siguientes limitaciones del circuito:

(*a*) R debe ser mayor de 5 kilohmios;

(*b*) R debe ser menor de $3.3 \times V$ megohmios (donde V es la tensión de alimentación).

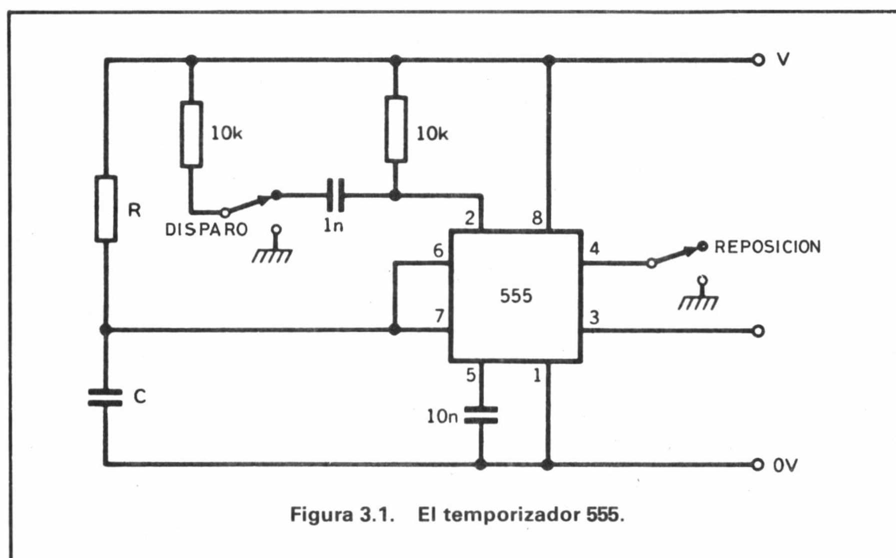


Figura 3.1. El temporizador 555.

Sería también conveniente que la resistencia y el condensador fueran fáciles de encontrar en el mercado. A tal efecto, es bueno realizar un repaso al campo de valores que dan el tiempo buscado.

He aquí el programa escrito teniendo en cuenta estos requisitos.

```
  10 PRINT "QUE PERIODO DE TIEMP
O EN MILI - SEGUNDOS NECESITA ";
  20 INPUT T
  30 PRINT "CUAL ES SU TENSION D
E ALIMENTACION EN VOLTIOS ";
  40 INPUT V
  50 LET A=T/(1.1*5)
  60 LET B=3.3*V*1000
  70 LET C=T/(1.1*B)
  80 PRINT "PARA UNA R MINIMA DE
 5K C DEBE  SER ";A;" MICROFARAD
IOS"
  90 PRINT "PARA UNA R MAXIMA DE
 ";B;"K C DEBE SER ";C;" MICROFA
RADIOS"
 100 PRINT
 110 PRINT "TECLEE UN VALOR DE R
 EN K OHMIOS PARA OBTENER EL VAL
OR DE C"
 120 INPUT D
 130 IF D<5 THEN PRINT "*ATENCIO
N:VALOR DE R DEMASIADO PEQUENO*"
 140 IF D>B THEN PRINT "*ATENCIO
N:VALOR DE R DEMASIADO GRANDE*"
```

```
 150 PRINT "EL VALOR DE C EN MIC
ROFARADIOS ES ";T/(1.1*D)
 160 GO TO 120
```

Las líneas 50, 60 y 70 son expresiones aritméticas directas que dan C para el valor más pequeño de R (5K) y el máximo valor de R y su C correspondiente, respectivamente. Las líneas 130 y 140 comprueban que el valor de la resistencia está comprendido en el margen recomendado; en caso contrario, se imprime un aviso.

Si se quiere imprimir una serie de valores de C correspondientes a valores de R comprendidos entre S y F hay que hacer las modificaciones siguientes. La mejor forma de entender este programa es la de imprimir los valores de J,J*I y S + J*I en cada pasada por el bucle.

```
 110 PRINT "VALOR INICIAL R (EN
K OHMIOS)";
 120 INPUT S
 130 PRINT "VALOR FINAL DE R (EN
K OHMIOS)";
 140 INPUT F
 150 PRINT "CUANTOS VALORES ";
 160 INPUT N
 170 LET I=(F-S)/(N-1)
 180 FOR J=0 TO N-1
 190 PRINT "PARA R= ";S+J*I;"K O
HMIOS C ES ";T/(1.1*(S+J*I));"MI
CRO FARADIOS"
 200 NEXT J
```

Preguntas de autoevaluación

1. Escríbase un programa que permita imprimir una tabla de conversión de pulgadas a milímetros cubriendo la gama de una pulgada a cinco pulgadas, con pasos de una décima de pulgada. Hay que partir del hecho de que una pulgada es aproximadamente 25 mm.

2. Modifíquese el programa de conversión de la pregunta **1** de manera que el usuario pueda especificar ahora la amplitud del margen y del paso.

3. En una resistencia se altera su valor con la temperatura según la fórmula

$$R = R0(1 + aT)$$

donde *R0* es la resistencia a cero grados centígrados, *a* es el coeficiente de temperatura y *T* es la temperatura de trabajo. Escríbase un programa que determine la resistencia correspondiente a una gama de temperaturas. El usuario puede especificar la resistencia a cero grados y el coeficiente de temperatura.

4. Una de las características menos atractivas del programa del temporizador 555 es que se imprimen los resultados en kilohmios cuando algunas veces sería más adecuado que vinieran en megohmios (es mejor poner 5.2 M que 5.200K). ¿Puede encontrar usted la forma de imprimir la resistencia siempre en la unidad correcta empleando sólo las instrucciones de BASIC vistas hasta ahora?

5. ¿Puede encontrar usted otra forma de escribir el bucle FOR en el programa modificado del temporizador 555, empleando STEP?

4

MATRICES Y CADENAS

El BASIC empieza a ser interesante cuando se domina la utilización de las matrices y de las cadenas. Si se lee un libro de programación avanzada se ve que un programa es solo un poco más que nuestra anterior definición de «un conjunto de instrucciones de BASIC» pero que está constituido por dos partes. Una de ellas se corresponde más o menos con nuestra definición: es simplemente un conjunto de instrucciones. La otra parte, se suele conocer como las «estructuras de datos». Son los elementos que el programa maneja. En este sentido, cabe relacionar las estructuras de datos con los «objetos» o «nombres» y las instrucciones BASIC con las «acciones» o «verbos».

Las únicas estructuras de datos vistas aquí hasta ahora son las variables simples y, por eso, los únicos programas que estamos en disposición de escribir son programas bastante sencillos. La matriz es la primera de las nuevas estructuras de datos que examinaremos, por dos razones: porque es fácil de entender y porque está normalizada de forma casi idéntica en todas las versiones de BASIC. Tras examinar las matrices, se verá la otra estructura de datos que tiene el BASIC: la cadena.

Las matrices

Hablando en términos generales, una matriz es un conjunto de variables simples. Si se tiene un conjunto de, por ejemplo, 10 va-

riables, lo más natural es hablar de la primera variable, de la segunda variable, etc. Este orden natural de las variables es la esencia de la matriz. Para facilitar las cosas, en BASIC todas las variables de una matriz tienen el mismo nombre, que es lo que se conoce como el nombre de la matriz. Así, por ejemplo, se puede llamar a un conjunto de 10 variables la matriz Z. La primera variable se llamaría Z(1), la segunda Z(2) y así sucesivamente hasta la Z(10). Obsérvese que se está hablando de 10 variables concretas de la Z(1) a la Z(10). Si, por accidente, se dijera Z(11) o Z(—1), la mayoría de las versiones BASIC, generarían un mensaje de error, con lo que no estarían diciendo que se está haciendo una tontería. Una variable de matriz se puede utilizar donde se pueda usar una variable simple. Las posibilidades de una matriz proceden del hecho de que el número utilizado para especificar la variable de matriz de que se está hablando puede ser cualquier expresión aritmética. Por ejemplo, Z(I), Z(A + 3) y Z(2*A + 1) son todas válidas.

La instrucción DIM

Todavía falta saber la forma de decir al microordenador la dimensión de una matriz concreta (para que pueda reservar suficiente espacio de memoria). Este problema se resuelve con la instrucción DIM (DIM viene de DIMensión). El formato de una instrucción DIM sencilla es

DIM nombre de variable (número de variables de la matriz)

Por ejemplo, la matriz Z de 10 variables podría haber sido declarada en la línea 10 así:

```
10 DIM Z(10)
```

La mayoría de las versiones de BASIC permiten especificar más de una matriz por instrucción DIM, por ejemplo DIM Z(10),A(25), pero aquí, a efectos de simplificación, se trabajará siempre con una matriz por instrucción DIM.

Otras cosas a recordar sobre las instrucciones DIM son: hay

que declarar una matriz antes de operar con ella y no se puede declarar la misma matriz más de una vez. Algunas versiones de BASIC permiten una cierta flexibilidad en estas condiciones (por ejemplo, el BASIC del ZX81 permite redefinir una matriz siempre y cuando se acepte perder los datos de la matriz anterior) pero es más seguro cumplirlas. Una cosa que algunas versiones de BASIC permiten hacer y que es demasiado importante para pasarla por alto, es la posibilidad de especificar el tamaño de una matriz mediante una expresión aritmética. Por ejemplo, DIM Z(N) o DIM Z(N + 36) son aceptables ambas en suficientes versiones de BASIC como para que se pueda decir que son casi estándar.

En programación general las matrices son muy prácticas para dos aplicaciones: la más común es el almacenamiento de un conjunto de números para uso posterior; y otra menos conocida es el trabajo con tablas de datos. A continuación se da un ejemplo de almacenamiento y clasificación de datos.

Clasificación por el método de la burbuja

Supóngase que se tiene un conjunto de datos y que se quiere clasificarlos por orden descendente. A tal efecto se podría usar el programa escrito a continuación.

```
  10 PRINT "CUANTOS DATOS";
  20 INPUT N
  30 DIM D(N)
  40 FOR I=1 TO N
  50 PRINT "NUMERO DE DATOS ";I;
" = ";
  60 INPUT D(I)
  70 NEXT I
 100 LET F=0
 110 FOR I=1 TO N-1
 120 IF D(I)>D(I+1) THEN GO TO 1
70
 130 LET T=D(I)
 140 LET D(I)=D(I+1)
 150 LET D(I+1)=T
 160 LET F=1
 170 NEXT I
 180 IF F=1 THEN GO TO 100
 190 PRINT "DATOS CLASIFICADOS"
```

Este programa es interesante por numerosas razones. La primera parte del programa, la que va desde la línea 10 a la 70, tiene

la función de introducir los datos en la matriz. Hay que destacar que las líneas 10 y 20 dan el número de datos, que sirve para definir una matriz suficientemente grande para admitir todos los datos de la línea 30. El bucle FOR de las líneas 40, 50, 60 y 70 introduce (INPUT) los datos en cada variable de la matriz. Adviértase el empleo de la I en la instrucción PRINT para decir de qué dato se trata y su empleo en la instrucción INPUT de la línea 60.

La segunda mitad del programa, líneas 100 a 190, clasifica los datos siguiendo un método conocido como de la burbuja. La clasificación de burbuja se basa en el examen de los datos por parejas. Si la primera variable de la pareja es mayor que la segunda no pasa nada. En cambio si la primera variable es menor que la segunda es que están en orden equivocado (recuérdese que el objetivo a alcanzar es poner los datos más altos en los primeros lugares de la matriz). La solución en tal caso es bien sencilla: intercambiar posiciones. Si se sigue haciendo esto (empezar por el principio de la línea e ir intercambiando los contenidos de las variables que están «mal» colocados) llegará un momento en el que los datos de la matriz estarán en el orden correcto. Se sabe que se ha acabado cuando se ha recorrido toda la matriz comparando todas las parejas consecutivas y no se ha hecho ningún intercambio porque todas están ya en el orden correcto.

Las líneas 110 a 170 exploran la matriz repetidamente. La línea 120 compara las dos variables de matriz, la D(I) y la D(I + 1). (Obsérvese que la posibilidad de utilizar expresiones aritméticas para especificar las variables de la matriz permite llamar a una variable de la matriz D(I) y a la siguiente D(I + 1)). Si el orden de los dos datos es el correcto no hay ningún intercambio y se llega a la GOTO 170, el final del bucle FOR. Si es preciso hacer un intercambio, las líneas 130, 140 y 150 se ocupan de ello. Primero guardan en una variable temporal T en memoria el dato D(I). Después D(I) se iguala a D(I + 1) y D(I + 1) se iguala a T. Si no se hubiera utilizado la variable temporal para guardar D(I), se habría perdido su dato cuando se iguala a D(I + 1). Hacer el intercambio no es pues tan fácil.

Otra aplicación interesante de las variables es la que pone al descubierto la variable F. Para saber cuando se ha acabado la clasificación es preciso detectar que no ha habido ningún intercambio

durante el bucle FOR. Si se pone F a cero antes de comenzar el bucle FOR y se pone a 1 si ha habido intercambio (se hace en la línea 160), esto puede servir para que la línea 180 decida si se ha llegado o no al «final de la clasificación». Si F=1 ha habido un intercambio y hay que explorar la matriz otra vez y se pasa GOTO 100. Si F=0 no ha habido ningún intercambio y la matriz está ya clasificada. Cuando se utiliza una variable en este sentido para indicar que algo ha sucedido o que no ha sucedido, recibe el nombre de señalizador (flag).

El programa todavía no está acabado. Se deja a modo de ejercicio la escritura de la parte del programa que saque impresa la matriz clasificada. Una última palabra sobre la clasificación por el método de la burbuja: no es la mejor forma de clasificar cosas. Se pueden encontrar métodos más rápidos en cualquier libro sobre programación pero ninguno es tan fácil de entender.

Tablas para almacenar características de componentes electrónicos

Todo el mundo está familiarizado con las tablas de obtención de datos. Cuando se tienen problemas con las operaciones aritméticas, concretamente con la multiplicación, la solución consiste en escribir un conjunto de tablas de multiplicar; es una de las formas más sencillas de tabla de obtención de datos. Para ilustrar este punto he aquí un programa para la tabla de obtención de datos de la multiplicación por dos.

```
10 DIM M(12)
20 FOR I = 1 TO 12
30 M(I) = 2*I
40 NEXT I
50 PRINT "2 VECES";
60 INPUT J
70 PRINT " =";M(J)
80 GOTO 50
```

Aunque el programa no tiene demasiada utilidad, muestra lo que se puede hacer con una tabla de obtención de datos. Las líneas de la 10 a la 40 cargan la tabla con los valores correctos y las

líneas de la 50 a la 80 hacen los cálculos. En caso de encontrar dificultades para entender el programa, la clave es recordar que la respuesta correspondiente a 2*J está almacenada en M(J).

Hay dos razones para utilizar una tabla de obtención de datos en vez de calcular directamente la respuesta. En primer lugar, porque la tabla de obtención de datos suele ser más rápida que el cálculo de la respuesta cada vez. (Por ejemplo, si en vez de la tabla de multiplicar por dos se tratara de la tabla de senos y cosenos). Y en segundo lugar, porque es posible que no haya una fórmula que dé los resultados almacenados en la tabla. Por ejemplo, no hay una fórmula que dé el número de puntos y rayas del Código Morse correspondientes a los dígitos 1 al 10, y una tabla de obtención de datos es muy útil para esta aplicación. Y se da con frecuencia el caso de que los datos de componentes electrónicos vienen en forma de gráficas y la única forma de utilizarlos es almacenarlos en una tabla de obtención de datos.

La mayoría de los principiantes tienen problemas con las tablas de obtención de datos cuando la «entrada» a la tabla no es un simple número entero. Por ejemplo, en el caso de una tabla de obtención del seno, el ángulo estaría comprendido entre 0 y 90 grados pero la búsqueda se ha de hacer en términos de 1 a N. La solución a este tipo de problemas no es demasiado difícil. Basta «trocear» la gama de datos de entrada en N trozos. En el caso de los senos se podría escoger N = 9 y obtener M(1) si el ángulo estuviera comprendido entre 0 y 10 grados, M(2) si estuviera comprendido entre 10 y 20 grados y así sucesivamente, hasta llegar a M(9) para ángulos comprendidos entre 80 y 90 grados. El paso de un ángulo en grados a un número entero (entre 1 y 9) se puede hacer mediante una serie de instrucciones IF o mediante un cálculo sencillo:

```
J = INT(X/10) + 1
```

donde X es el ángulo e INT es una función de BASIC nueva que pasa un número real a número entero eliminando la parte decimal (INT(3.445) = 3). Hay que tener cuidado cuando se utiliza este método porque si X = 89, daría J = 9, pero si X = 90, daría J = 10, que es mayor que el tamaño de la matriz. En otras palabras, ¡cuidado con los números finales!

Las dimensiones de las matrices

La mayoría de las tablas de datos que se usan normalmente son un poco más complicadas que las consideradas hasta ahora. Por ejemplo, si se quiere saber la distancia por carretera entre dos ciudades, la tabla tendría dos entradas: el nombre de la primera ciudad y el nombre de la segunda. Si se observa la tabla se comprueba que es obvio que se amplíe en dos direcciones, una correspondiente a la ciudad de salida y la otra a la ciudad de llegada. Se dice, en tal caso, que la tabla es «bi-dimensional». Esta idea es aplicable a las matrices de BASIC. Una matriz de dos dimensiones es un conjunto de variables que tienen todas el mismo nombre y están especificadas por dos números. Ejemplos de matrices con dos dimensiones son D(3,3) y M(1,2). Una matriz bi-dimensional se define con la instrucción DIM.

DIM nombre de matriz (tamaño máximo de la matriz 1, tamaño máximo de la matriz 2)

Por ejemplo, la matriz B, de 10 por 10 variables, se declararía mediante la instrucción

DIM B (10,10)

Si se declara la matriz D como de M por N elementos, es decir con la instrucción DIM D(M,N), el número total de variables así creadas es M*N, que puede ser un número mayor de lo esperado. La pequeña e inocente matriz B, de tamaño 10 por 10, es de 100 variables. Empezando por la variable M(1,1) y acabando por la M(10,10), se pueden enumerar 100 variables diferentes. Inténtese hacerlo a modo de ejercicio. El asunto es que una matriz de dos dimensiones puede llenar rápidamente todo el espacio de memoria disponible.

Algunos ordenadores permiten definir matrices de tres, cuatro, cinco o más dimensiones. Estas matrices no se suelen utilizar con tanta frecuencia como las matrices de una y dos dimensiones y necesitan mucha más memoria.

Un selector de resistencias

A título de ejemplo práctico, el siguiente programa selecciona una resistencia adecuada a nuestras necesidades entre las de la gama estándar de valores. Al calcular el valor de la resistencia a poner en un circuito concreto muchas veces se acaba con números tan anormales como el 45943. El problema es que no se pueden comprar resistencias de 45943 ohmios porque los fabricantes de resistencias sólo fabrican resistencias de ciertos valores, los de las llamadas gamas E12 y E24. La gama de valores E12, para cualquier resistencia buscada, ofrece un valor E12 preferente que está comprendido en la gama de más o menos el 10 por ciento del valor de aquella. Paralelamente, los valores E24 ofrecen una tolerancia del 5 por ciento de cualquier valor. Inteligente ¿no?

El programa dado a continuación pide el valor calculado y luego busca en la tabla E12 el valor preferente más aproximado y saca impreso el valor junto con el porcentaje de error en que se está incurriendo al adoptarlo. (Obsérvese que este porcentaje de error debe ser por definición menor al 10 por ciento en la serie E12).

```
  10 PRINT "SELECTOR DE RESISTEN
CIAS"
  20 DIM R(13)
  30 LET R(1)=1
  40 LET R(2)=1.2
  50 LET R(3)=1.5
  60 LET R(4)=1.8
  70 LET R(5)=2.2
  80 LET R(6)=2.7
  90 LET R(7)=3.3
 100 LET R(8)=3.9
 110 LET R(9)=4.7
 120 LET R(10)=5.6
 130 LET R(11)=6.8
 140 LET R(12)=8.2
 144 LET R(13)=10
 145 LET S=1
 150 PRINT "VALOR CALCULADO DE L
A RESISTENCIA EN OHMIOS = ";
 160 INPUT V
 170 IF V<10 THEN GO TO 210
 180 LET V=V/10
 190 LET S=S*10
 200 GO TO 170
 210 LET J=1
 220 LET M=R(1)-V
 230 LET M=ABS (M)
 240 FOR I=2 TO 13
```

```
250 LET D=R(I)-V
260 LET D=ABS (D)
270 IF D<M THEN LET J=I
280 IF D<M THEN LET M=D
290 NEXT I
300 PRINT "VALOR PREFERENTE MAS
CERCANO = ";R(J)*S
310 PRINT "PORCENTAJE DE ERROR
= ";(V-R(J))/V*100;" % "
320 GO TO 145
```

El que haya seguido el libro hasta aquí encontrará el programa muy elemental por lo que se deja al lector el ejercicio de mejorarlo. Dos posibles mejoras consisten en ampliarlo también a la serie E24 y en sacar impresos dos valores cuando hay dos valores posibles. Los únicos puntos que merece la pena explicar son la limitación de V a números menores de 10 en las líneas 170 a 200 y la forma de usar S para recordar cuantas veces se ha dividido por diez. Las líneas 210 a 290 hallan la diferencia más pequeña entre V y la tabla de valores E12. En las líneas 230 y 260 se hace uso de otra función BASIC nueva, la ABS, que saca la diferencia absoluta. La función ABS se limita a olvidar el signo y hace todos los números positivos. Si se estudia el programa detenidamente el resto no ha de ofrecerle dificultades. Vea si puede responder a las siguientes preguntas:

(a) ¿Por qué se hace J = 1 en la línea 210?
(b) ¿Qué hace la línea 270?
(c) ¿Funcionaría el programa si se intercambiaran las líneas 270 y 280?

Las cadenas

Cuando se llega al tema de las cadenas se encuentra el primer problema. No es que las cadenas sean más difíciles de entender que las matrices, es que su implementación difiere de un BASIC a otro. El problema está en que tres versiones BASIC, muy populares las tres, operan con cadenas de forma muy diferente. El BASIC Microsoft (el de la mayoría de los micros como Apple, Pet, etc., véase el Capítulo 1 para más detalles), el BASIC del ZX Spectrum y el BASIC del ZX81 operan cada uno de ellos de forma peculiar al

utilizar cadenas. De los tres, el Microsoft es el más extendido y el considerado como BASIC estándar por lo que aquí se tratarán las cadenas a la forma del BASIC Microsoft pero, para no dejar a los usuarios del ZX Spectrum y del ZX81 en situación desventajosa, se explicará la forma de pasar de Microsoft a las versiones de estos microordenadores.

¿Qué es una cadena?

Ya nos hemos encontrado con cadenas en las instrucciones PRINT. Por ejemplo:

```
PRINT "ESTA ES UNA CADENA LITERAL"
```

Tal como se explicó en el Capítulo 2, los caracteres encerrados entre comillas forman una cadena literal (de letras). Una cadena no es más que un conjunto de caracteres, una «cadena» de caracteres. La mayoría de las versiones BASIC exigen que las cadenas estén encerradas entre comillas pero hay que tener en cuenta que las comillas no forman parte de la cadena.

Obviamente, lo que falta ahora es una forma de guardar cadenas para su posterior utilización. Necesitamos las variables que puedan contener cadenas, las variables de cadena. La mayoría de las versiones BASIC utilizan el signo $ para indicar que una variable puede contener una cadena. Por ejemplo, A$ es una variable de cadena al igual que A es una variable numérica. Se pueden utilizar A y A$ en el mismo programa porque ambas tienen significados completamente distintos.

Expresiones de cadenas

Como en el caso de las variables numéricas, también se puede asignar un valor a una variable de cadena. La única diferencia es que el valor es una cadena.

Por ejemplo:

```
10 A$ = "MI NOMBRE"
```

asigna la cadena «MI NOMBRE» a la variable A$. Pero si se intenta poner

```
A = "MI NOMBRE"
```

se recibirá un mensaje de error porque la A, sin el signo del dólar, es, por definición una variable numérica. En forma similar,

```
A$ = 234
```

provocará un mensaje de error porque la cadena no está encerrada entre comillas. Sólo se puede asignar una cadena a una variable de cadena

```
A$ = "234"
```

es aceptable, sin embargo, porque «234» es una cadena.

Si se añade la línea

```
20 PRINT A$
```

a este ejemplo y se pasa el programa resultante, veremos

```
MI NOMBRE
```

en la pantalla. Obsérvese que las comillas han desaparecido porque las comillas no forman parte de la cadena. Sólo sirven para marcar dónde empieza y dónde acaba.

Las variables de cadena se pueden incluir en las instrucciones PRINT e INPUT de la misma forma que las variables numéricas. Por ejemplo,

```
10 INPUT A$
20 PRINT "CADENA = ",A$
```

es del todo correcto.

Ya podemos guardar cadenas para su posterior utilización pero, por el momento, no conocemos ninguna forma de modificar cadenas.

El método más sencillo de modificar cadenas viene bajo el muy extravagante nombre de «concatenación» que sencillamente significa poner dos cadenas juntas (concatenarlas). El símbolo normalizado para la concatenación es + pero no por ello se ha de creer que tiene algo que ver con la adición. La concatenación se comprende más fácilmente a través de un ejemplo:

```
10 PRINT "CUAL ES SU APELLIDO";
20 INPUT A$
30 PRINT "CUAL ES SU NOMBRE";
40 INPUT N$
50 C$ = N$ + A$
60 PRINT "SU NOMBRE COMPLETO ES";C$
```

Al pasar el programa, éste preguntará el apellido y el nombre y sacará impreso el nombre completo en el orden correcto. El único problema es que no deja espacio en blanco entre los dos. La mayoría de las versiones BASIC permiten solventar este problema con la línea

```
50 C$ = N$ + " " + A$
```

Pero ¡cuidado! porque algunas versiones BASIC solo permiten una + por instrucción. En estas versiones habría que utilizar

```
50 C$ = N$ + " "
51 C$ = C$ + A$
```

Instrucciones IF con cadenas

Las cadenas no han sido de mucha utilidad hasta el momento. Lo único que se puede hacer con ellas, por ahora, es introducirlas, imprimirlas y juntarlas. La mayoría de las veces las cadenas sirven para guardar una respuesta a una pregunta y luego decidir qué significa la respuesta. Considérese por ejemplo lo siguiente:

```
10 PRINT "QUIERE PASAR ESTE PROGRAMA" (SI = 1,
   NO = 0)";
20 INPUT A
30 IF A = 0 THEN STOP
```

El microordenador hace la pregunta en castellano pero las únicas respuestas que puede aceptar son números. Sería mucho mejor si la respuesta pudiera ser SI o NO. Esto es posible utilizando las cadenas en instrucciones IF. Por ejemplo,

```
10 PRINT "QUIERE PASAR ESTE PROGRAMA (RESPON-
   DA SI O NO)"
20 INPUT A$
30 IF A$ = "NO" THEN STOP
```

Las dos únicas relaciones que es posible comprobar con la igualdad (por ejemplo A$ = "NO") y la desigualdad (por ejemplo A$ < >"SI"). La razón de esto hay que buscarla en que las relaciones «mayor que» y «menor que» no tienen un significado normalizado en el campo de las cadenas. Algunas versiones de BASIC permiten su uso pero lo que hacen varía de unas a otras. (Ya se verá que los signos > y < pueden ser utilizados en cadenas de solo una letra y que es posible comparar la longitud de las cadenas). En el ejemplo anterior la respuesta NO parará el programa y la respuesta SI permitirá su continuación, pero lo mismo harían las respuestas S, STOP, OK, o cualquier otra cadena que se teclee. El problema está en que el programa comprueba si la respuesta es NO pero no mira a ver si es SI. Para que hiciera la comprobación para el SI, habría que añadir la línea

```
40 IF A$< > "SI" THEN GOTO 10
```

Las funciones de cadenas

Igual que cabe la posibilidad de juntar cadenas utilizando el signo más, tal como se ha visto, podría pensarse que lo único que hay que hacer es usar el signo menos para «quitar una cadena de otra» pero las cadenas no son como los números. Lo que puede ocurrir es que sea necesario coger los primeros caracteres de una cadena y ponerlos en otra. Esto es difícil de escribir de una forma parecida a la aritmética. De ahí que se utilicen funciones de cadenas en vez de símbolos. Las funciones de cadena son similares a las funcio-

nes matemáticas del tipo SIN(X). La función SIN(X) coge el número X y calcula el seno correspondiente que es también un número. Una función de cadena coge una cadena y calcula una respuesta que es también una cadena. Tal como ha quedado dicho, el problema con las funciones de cadena es que no están normalizadas. Aquí se verán, en primer lugar, las versiones Microsoft.

LEFT$(A$,n)

da los n caracteres de la cadena A$ situados a la izquierda. Por ejemplo,

```
PRINT LEFT$("ABCDE",3)
```

imprime ABC.

RIGHT$(A$,n)

da los n caracteres de la cadena A$ situados a la derecha. Por ejemplo,

```
PRINT RIGHT$("ABCDE",2)
```

imprime DE.

MID$(A$,n,m)

da n caracteres empezando por el enésimo. Por ejemplo,

```
PRINT MID$("ABCDE",2,2)
```

imprime BC.

Hay más funciones de cadena que el BASIC acepta pero éstas son las más comunes y las más útiles.

Hay otra función relativa a las cadenas que es preciso conocer. Se diferencia de las tres anteriores en que no da una cadena como respuesta sino un número. Es la

LEN(A$)

Esta función da el número de caracteres de la cadena. Por ejemplo,

```
PRINT LEN("ABCDE")
```

imprime 5.

Utilización de las funciones de cadenas

La mejor forma de comprender las funciones de cadena es utilizarlas. La dificultad más común que tienen los programadores que empiezan a utilizar funciones de cadenas es emplearlas para resolver operaciones sencillas que necesitan más de una cadena. Por ejemplo, ¿cómo sacar impresa una cadena letra por letra? Así:

```
10 INPUT S$
20 FOR I = 1 TO LEN(S$)
30 PRINT MID$(S$,I,1)
40 NEXT I
```

Este método puede servir para explorar una cadena a la búsqueda de un carácter determinado.

Otro problema es cómo imprimir una cadena eliminando las primeras n letras.

Se hace así:

```
10 INPUT S$
20 PRINT RIGHT$(S$,LEN(S$)—N)
```

Para comprender el funcionamiento de este programa hay que tener en cuenta que si no se quiere que se impriman los primeros caracteres N lo mejor es imprimir los últimos LEN(S$) — N caracteres.

Un problema más difícil es insertar una cadena en otra. Si quisiéramos insertar la cadena A$ en la cadena S$ tras el enésimo carácter, se haría así:

```
10 INPUT A$,S$,N
20 A$ = LEFT(A$,N) + S$ + RIGHT$(A$,LEN(A$) — N)
30 PRINT A$
```

La expresión de la línea 20 es preocupante. Si se la observa detenidamente se verá que la primera parte extrae los N primeros caracteres y la última parte extrae el resto de la cadena. Tras la descomposición de la cadena A$ en estas partes, el operador de concatenación con la cadena S$ en medio.

Las versiones BASIC del ZX81 y del ZX Spectrum

Como se ha mencionado anteriormente, el ZX81 y el ZX Spectrum tratan las cadenas de una forma distinta que el BASIC Microsoft. En algunos casos, los métodos Sinclair son más sencillos y más potentes que las RIGHT$, LEFT$ y MID$.

En vez de utilizar funciones de cadena el ZX81 emplea una notación llamada «segmentación» para describir la parte de la cadena en la que estamos interesados. Esta notación es

```
cadena (N TO M)
```

y significa «la parte de la cadena comprendida entre el carácter N y el carácter M». Por ejemplo, "ABCDEF"(3 TO 5) es CDE. Lo único que complica un poco esta notación es el número de cosas que se dejan fuera de la misma. Si se deja de poner N se supone N = 1 y si se deja de poner M se da por supuesta toda la longitud de la cadena. Se puede también dejar de poner la TO y el resultado es el mismo que si se hubiera escrito (N TO N). Unos pocos ejemplos servirán de ayuda:

```
"ABCDEF" (  TO 3) = "ABCDEF" (1 TO 3) = "ABC"
"ABCDEF" (3 TO  ) = "ABCDEF" (3 TO 6) = "CDEF"
"ABCDEF" (  TO  ) = "ABCDEF" (1 TO 6) = "ABCDEF"
"ABCDEF" (3)      = "ABCDEF" (3 TO 3) = "C"
```

Hay que recordar que N y M deben referirse a una parte de la cadena que exista a menos que N sea mayor que M. Por ejemplo, "ABC"(2 TO 4) da un error pero "ABC"(4 TO 2) da una cadena nula (se explica más adelante).

Formar las notaciones de «segmentación» equivalentes a las funciones de cadena Microsoft no deberían plantear ya ningún problema

al lector. Para LEFT$(A$,N) hay que usar A$(1 TO N) o A$(TO N); para RIGHT$(A$,N) hay que usar A$(LEN(A$)—N TO) y para MID$(A$,N,M) hay que usar A$(N TO N + M). En realidad, es bastante fácil.

Si se emplea este método es fácil convertir cualquier programa, con funciones Microsoft, a la notación de «segmentación». No obstante, el ZX81 ofrece más pues la parte de una cadena especificada por el segmentador no solo puede ser extraída y utilizada, sino que también puede ser asignada. Por ejemplo,

```
10 A$ = "ABCDEF"
20 A$(2 TO 4) = "******"
30 PRINT A$
```

imprime A***EF. Obsérvese que la parte de la cadena especificada por el segmentador (2 TO 4) ha sido sustituida por la cadena de asteriscos y que sólo se han utilizado los tres primeros asteriscos.

El BASIC del Atom

Se puede decir en términos generales que el microordenador Atom hace las cosas de una forma tan distinta que el mejor consejo a dar a los usuarios de Atom es que lean detenidamente sus manuales. El único peligro que hay aquí es que la programación en BASIC Atom provoca un aislamiento con respecto al resto de la comunidad BASIC. Esto se puede comprobar leyendo el resto de este libro y familiarizándose con la forma Microsoft.

El método Atom de tratamiento de las cadenas es un poco difícil de explicar a la luz de lo aprendido hasta aquí. La diferencia básica es la forma de usar el nombre de una cadena. En el BASIC de Atom el nombre de una cadena hace las funciones de «puntero» del principio de la cadena. El Atom pide también que se le diga el tamaño máximo que tendrá cada cadena y ello antes de operar con ella. En forma resumida, las diferencias son:

(a) Las cadenas llevan nombres como $A en vez de A$.
(b) Para definir la máxima longitud de cada cadena hay que dimensionar una matriz del mismo nombre.

(c) La operación + no actúa. En vez de A$ = B$ + C$ hay que usar $A = $B $A + LEN(A) = $C.
(d) Para A$ = RIGHT$(B$,N) hay que usar $A = $B + LEN(B) —N.
(e) Para A$ = LEFT$(B$,N) hay que usar $A = $B $A + N = "".
(f) Para A$ = MID$(B$,N,M) hay que usar $A = $B + N$A + M = "".
(g) Para LEN(A$) hay que usar LEN(A).

La cadena nula

Es el momento adecuado para presentar una cadena muy especial: la cadena nula. La cadena nula juega el papel del cero en las funciones de cadenas. Es una cadena que no contiene ningún carácter. Concretamente A$ = "" es la cadena nula, y no hay que confundirla con B$ = " ", que es una cadena que contiene un solo espacio en blanco. La diferencia se pone de manifiesto en su longitud, porque LEN(A$) es cero y LEN(B$) es uno.

Números a cadenas y cadenas a números

Se apuntó anteriormente que A$ = 234 no es aceptable porque 234 no es una cadena. ¿Y si se quieren usar números como parte de una cadena? La solución es la utilización de la función STR$(X) con la cual se obtiene la cadena que podría producirse si se imprimiera X.

Por ejemplo:

```
10X = 354.56
20A$ = STR$(X)
30 PRINT A$,X
```

imprimiría 354.56 dos veces. Esta solución puede parecer no muy útil pero, para programación avanzada, puede resultar valiosísima. Por ejemplo, si se quiere imprimir un número solo en el caso de que tenga que situarse en un determinado espacio, se podría utilizar

```
10 IF LEN(STR$(X)) < 4 THEN PRINT X
```

En sentido contrario, el paso de una cadena de números a un valor es igual de fácil. La función VAL(A$) da el valor numérico de la cadena A$. Por ejemplo, X = VAL("234") pone la cantidad doscientos treinta y cuatro en la variable X.

Los caracteres y su orden

A ningún lector le parecerá un problema responder si se le pregunta cuál es la primera letra del alfabeto. De la misma forma que las personas ordenan el alfabeto, el ordenador clasifica todos los símbolos que puede imprimir. Por desgracia este orden no es el mismo en todos los microordenadores, por lo que hay que tener cuidado. Una función muy útil es la CHR$(N), que proporciona el carácter que ocupa la enésima posición en la clasificación del microordenador. Para saber el orden que sigue un equipo concreto se puede utilizar este programa:

```
10 FOR I = 0 TO 255
20 PRINT "EL CARACTER";I, "ES", CHR$(I)
30 FOR J = 1 TO 300
40 NEXT J
50 NEXT I
```

Este programa imprimirá todos los caracteres uno por uno. El bucle FOR J de las líneas 30 a 50 simplemente deja pasar tiempo con el fin de que se puedan ver los caracteres que van saliendo impresos. Si salieran en la pantalla algunos espacios en blanco inesperados, no debe olvidarse que algunos caracteres no se pueden imprimir. Son cosas que no se pueden ver, como el espacio en blanco y el retroceso.

Una vez conocido el orden que adopta la máquina, se puede volver a un tema tratado anteriormente. Si se quieren clasificar alfabéticamente algunos nombres se ha de partir de la posibilidad de comparar los lugares que ocupan las letras en ese orden alfabético. Esto se puede hacer utilizando las relaciones «mayor que» y «menor que» en una instrucción IF. Por ejemplo,

```
10 INPUT A$
20 INPUT B$
```

```
30 IF A$< B$ THEN PRINT A$; "SITUADO ANTES QUE";B$
40 IF B$< A$ THEN PRINT B$; "SITUADO ANTES QUE";A$
```

Si se teclea Z en respuesta a la primera INPUT y después X en respuesta a la segunda INPUT, el programa debe imprimir X SITUADO ANTES QUE Z. En el caso de cadenas de un solo carácter es mejor llamar a la relación «menor que» como «situado antes que» y a la «mayor que» como «situado después que». En el caso de «menor o igual que» o «igual que» el paso es obvio. Lo que ocurre si se intenta aplicar las relaciones < ó > a cadenas de más de un carácter depende de la versión BASIC que se utilice. Lo mejor es evitar este caso pero si alguien quiere saber qué es lo que pasa puede usar el programa anterior para averiguar cómo resuelve este problema su versión de BASIC.

Matrices de cadenas

Después de haber comprobado ya lo útiles que son las matrices, es bueno saber que es posible trabajar con matrices de cadenas además de con las matrices de números. La idea de la matriz de cadenas es muy similar a la de la matriz normal. Hay que decir al microordenador cuántos elementos tiene la matriz y esto se consigue con la instrucción DIM.

DIM variable de cadena (número de elementos)

Por ejemplo, una matriz de 10 cadenas se declararía así:

```
10 DIM A$(10)
```

Un ejemplo sencillo ayudará a explicar el tema de las matrices de cadenas. Supóngase que se quiere guardar una lista de los colores favoritos.

```
10 DIM C$(5)
20 FOR I = 1 TO 5
30 PRINT "NOMBRE UNO DE SUS COLORES
   FAVORITOS";
```

```
40 INPUT C$(I)
50 NEXT I
60 PRINT "LUEGO SUS COLORES FAVORITOS SON —"
70 FOR I = 1 TO 5
80 PRINT I;"...";C$(I)
90 NEXT I
```

Obsérvese que una característica importante de una matriz de cadenas es que hay un orden de los elementos. Es decir, si se quiere saber el primer color se examina el elemento C$(1).

Algunos microordenadores permiten incluso disponer de matrices de cadenas multidimensionales y con esta observación queda dicho casi todo lo que hay que decir sobre las matrices de cadenas excepto que no son necesarias con mucha frecuencia y eso es bueno, porque exigen una cantidad de memoria alarmante.

Un programa para la práctica del Código Morse

Es sorprendente descubrir que en estos tiempos de la electrónica avanzada sigue estando en vigor el Código Morse. Hay que admitir en su favor que, en condiciones difíciles, una simple señal Morse llegará nítidamente hasta allí donde las señales vocales pueden, en el mejor de los casos, resultar casi ininteligibles. Además los transmisores de Morse son muy sencillos y ocupan muy poca anchura de banda. Con todas estas ventajas no debe sorprender que una de las condiciones para obtener algunas de las licencias de radioaficionado sea mostrar un cierto dominio del lenguaje Morse.

Lo que viene a continuación, el programa más largo de este libro hasta ahora, para la práctica del Código Morse. Hace un uso extensivo de las cadenas pero no por ello es difícil de pasar a las condiciones particulares de cualquier microordenador.

```
10 DIM C$(26)
15 LET S=0
20 LET C$(1)=". -"
30 LET C$(2)="- . . ."
40 LET C$(3)="- . - ."
50 LET C$(4)="- . ."
60 LET C$(5)="."
70 LET C$(6)=". . - ."
80 LET C$(7)="- - ."
```

```
  90 LET C$(8)=". . . ."
 100 LET C$(9)=". ."
 110 LET C$(10)=". - - -"
 120 LET C$(11)="- . -"
 130 LET C$(12)=". - . ."
 140 LET C$(13)="- -"
 150 LET C$(14)="- ."
 160 LET C$(15)="- - -"
 170 LET C$(16)=". - - ."
 180 LET C$(17)="- - . -"
 190 LET C$(18)=". - ."
 200 LET C$(19)=". . ."
 210 LET C$(20)="-"
 220 LET C$(21)=". . -"
 230 LET C$(22)=". . . -"
 240 LET C$(23)=". - -"
 250 LET C$(24)="- . . -"
 260 LET C$(25)="- . - -"
 270 LET C$(26)="- - . ."
 300 PRINT "APARECERA UN SIMBOLO
 MORSE - PULSE LA TECLA CORRESPO
NDIENTE"
 310 PRINT "TIENE 3 OPORTUNIDADE
S PARA DAR LA RESPUESTA CORRECTA
"
 320 PRINT "ANTES DE QUE EL ORDE
NADOR DE LA RESPUESTA CORRECTA"
 330 PRINT
 340 PRINT "ESTA LISTO (SI/NO)"
 350 INPUT A$
 360 IF A$="NO" THEN STOP
 370 IF A$<>"SI" THEN GO TO 300
 380 FOR I=1 TO 10
 390 LET C=INT (RND(0)*25+1)
 400 LET J=0
 410 LET J=J+1
 420 PRINT "QUE ES ";C$(C)
 430 INPUT A$
 440 IF A$=CHR$ (64+C) THEN GO T
O 500
 450 PRINT "*** ERROR *** INTENT
ALO OTRA VEZ"
 460 IF J<3 THEN GO TO 410
 470 PRINT "EL SIMBOLO REPRESENT
A A LA ";CHR$ (64+C)
 480 NEXT I
 490 GO TO 530
 500 PRINT "CORRECTO !"
 510 LET S=S+1
 520 NEXT I
 530 PRINT
 540 PRINT "SU MARCADOR ES ";S;"
/10"
 550 IF S<5 THEN PRINT "SIGA PRA
CTICANDO"
 560 IF S>4 THEN PRINT "MUY BIEN
"
 570 IF S>7 THEN PRINT "NO LO PO
DIA HABER HECHO MEJOR!"
 580 IF S=10 THEN PRINT "LO HA A
CERTADO TODO!"
```

Las líneas 10 a la 270 definen una matriz C$ que contiene los símbolos Morse de la A a la Z. El símbolo correspondiente a la A está almacenado en C$(1), el de la B está almacenado en C$(2) y así sucesivamente hasta la Z, en C$(26). Las líneas 300 a 330 imprimen algunas indicaciones (que cada uno puede mejorar) y las líneas 340 a 370 preguntan al usuario si está listo para empezar. Las líneas 380 a 530 constituyen la parte del programa que hace casi todo el trabajo. Son seleccionados aleatoriamente diez símbolos. El bucle FOR que comienza en la línea 380 hace que todo sea repetido diez veces y la instruccion de la línea 390 selecciona un símbolo aleatoriamente.

La línea 390 exige una explicación. La función RND(0) es una función que genera números aleatorios comprendidos entre el 0 y el 1. (Compruebe que ésta funciona porque no siempre es estándar). Una vez obtenido un número comprendido entre el 0 y el 1, multiplicándolo por 25 se obtiene un número comprendido entre el 0 y el 25, y sumándole una unidad se obtiene un número comprendido entre el 1 y el 26. La función INT sirve para convertir este número en un entero y elimina la parte decimal. Por ejemplo, INT(3.14159) es sencillamente 3. A estas alturas debe estar suficientemente claro lo que hace la línea 390: proporciona un número aleatorio entero comprendido entre el 1 y el 26. Este número es almacenado en C y utilizado para seleccionar qué símbolo Morse será presentado en pantalla en la línea 420.

J, puesto a cero en la línea 400, sirve para llevar la cuenta del número de intentos hechos por el usuario. La respuesta del usuario es aceptada en la línea 430 y en la línea 440 es comparada con la respuesta correcta. La única dificultad que plantea la línea 440 es el uso de la función CHR$(64 + C). En la mayoría de los microordenadores el alfabeto comienza después de 64 caracteres que no pertenecen al alfabeto. La letra A es el carácter 65. En la matriz de símbolos Morse, C$, el símbolo para A, está almacenado en C$(1). Por tanto, para generar una A, partiendo de CHR$(I), I debe valer 65, es decir 64 + 1. En general, para producir el carácter correspondiente al código almacenado en C$(I), se utiliza la función CHR$(64 + 1). Está claro que no en todos los microordenadores será adecuada tal solución, porque depende del orden en que son generados los caracteres por la función CHR$(I). Por ejemplo,

CHR$(38) da la A en el ZX81, luego en este programa de Morse habría que usar la función CHR$(37 + I).

Si la respuesta dada fuera equivocada se iría a parar a las líneas 450 y 460, que imprimen un mensaje y comprueban si el usuario ha agotado ya las tres posibilidades de responder. Si se han hecho menos de tres intentos, se repite la pregunta volviendo a la línea 410. Si se han hecho más de tres intentos sale impresa la respuesta y es seleccionado otro símbolo, por efecto de la instrucción NEXT I. Si el bucle FOR ha terminado, es decir, se han presentado diez símbolos, se ejecuta la línea 490 y se salta a la línea 530.

Si la respuesta dada fuera la correcta, la línea 500 imprimiría un mensaje y la línea 510 sumaría una unidad al tanteo guardado en S. La siguiente NEXT I selecciona otra vez otro símbolo y si el bucle FOR se ha completado el control pasa a la línea 530. Las líneas 530 a 580 simplemente imprimen una especie de calificación del usuario en función de las respuestas correctas. Son fáciles de entender pero hay que tener en cuenta que para un tanteo concreto puede salir más de un mensaje de congratulación.

Son muchas las mejoras que se pueden introducir en este programa. Esta es una buena posibilidad para que cada uno introduzca los perfeccionamientos que crea más convenientes. Algo que haría que el programa pareciera más real, sería utilizar los sonidos que cada microordenador puede generar, para repetir la señal de Morse en vez de limitarse a presentarla en pantalla.

Preguntas de autoevaluación

1. Escríbase un programa que lea 10 números y luego los imprima en orden inverso.

2. Modifíquese el programa de la pregunta **1** de manera que invierta el orden de cualquier cantidad de números.

3. Escríbase un programa que dé entrada a una cadena y luego la imprima en orden inverso.

4. Escríbase un programa que acepte el código de colores de una resistencia y luego imprima su valor. El código de colores es:

Color	Número
Negro	0
Marrón	1
Rojo	2
Naranja	3
Amarillo	4
Verde	5
Azul	6
Violeta	7
Gris	8
Blanco	9

Los dos primeros colores de la resistencia indican las dos cifras más significativas y el tercero el número de ceros que vienen a continuación de la segunda cifra.

5

FUNCIONES, SUBRUTINAS Y EXPRESIONES

Las funciones y las subrutinas son muy importantes en BASIC porque, empleadas adecuadamente, acortan los programas y los hacen más fáciles de depurar. En cierto modo, las subrutinas son formas ampliadas de las funciones por lo que es lógico analizar las funciones en primer lugar.

Las funciones

En BASIC hay dos formas de funciones: las funciones introducidas por el usuario y las funciones incorporadas. De ambos tipos, las funciones incorporadas son las más conocidas y de hecho en los capítulos anteriores las hemos estado utilizando. Una función es, hablando en términos generales, una orden para que sean efectuados unos cálculos y salga una sola respuesta. Por ejemplo, SQR(4) hace que el microordenador calcule la raíz cuadrada de 4. La respuesta única es, en este caso, el 2. Como se produce solo una respuesta por función, se puede utilizar el signo igual para decir dónde almacenar esta respuesta para su uso posterior. Por ejemplo:

```
10 A = SQR(4)
20 PRINT A
```

imprimiría el 2. Hay que señalar que si una función diera más de un valor como respuesta, sería imposible utilizar el signo igual sin desarrollar una forma de seleccionar cual de las posibles respuestas habría de ser almacenada en la variable. Por esta razón la función SQR sólo da la raíz cuadrada positiva del número e ignora la negativa. Por lo tanto, aunque sabemos que hay dos raíces cuadradas de 4 (+2 y —2), esto es solo aplicable a los humanos.

Otra consecuencia de la característica de respuesta única de las funciones es su posibilidad de uso en expresiones aritméticas sin demasiadas restricciones. Por ejemplo:

```
10 A = (SQR(4)*3)+2
20 PRINT A
```

imprimiría el 8

La regla en vigor es: el BASIC evalúa una función (calcula la respuesta) y luego pasa a efectuar el resto de la expresión como si sustituyera la función por su respuesta. Así (SQR(4)*3) pasa a ser (2*3) y luego 6. Generalizando: las funciones tienen más prioridad que las otras operaciones aritméticas y se realizan en primer lugar.

El valor o los valores asignados a una función reciben el nombre de argumento o parámetros de la función. Por ejemplo, en SQR(X) X es el argumento. No es necesario que una función tenga solo un argumento aunque la mayoría sólo tienen uno. Otro lujo que la mayoría de las versiones de BASIC permiten es el empleo de expresiones en el interior de una función, es decir, los parámetros pueden ser expresiones aritméticas. Por ejemplo:

```
10 A = SQR(2*2)
20 PRINT A
```

imprimirá un 2. Los más osados pueden intentar hacer ésta:

```
10 PRINT SQR(SQR(SQR(2)))
```

El tipo de respuesta que pueden dar las funciones no está limitada a los números. Las funciones de cadena, por ejemplo, producen cadenas. Del mismo modo, los parámetros de las funciones pueden ser de muy diferentes tipos.

La mayoría de las versiones de BASIC incorporan una amplísima gama de funciones. Se puede decir que cuanto mejor es el BASIC más funciones tiene. Entre las funciones que vale la pena conocer se encuentran las siguientes:

ABS(X): brinda el valor absoluto de X, es decir, elimina el signo negativo si lo hay. ABS(2) es 2 y ABS(—2) es también 2.
COS(X): ofrece el coseno de X. El único punto que necesita aclaración es el tema de las unidades en que se expresa X. Todos los BASICs que conozco exigen que X esté en radianes. Para convertir los grados a radianes hay que multiplicar por 180 y dividir por 3.14159.
EXP(X): da e elevado a X. La función EXP aumenta de valor muy rápidamente hasta el punto de que el valor máximo de X que puede emplearse no suele ser mucho mayor de 80.
INT(X): brinda el número entero mayor más pequeño o igual a X. INT(1.5) es 1 pero INT(—1.5) es —2 porque —2 es *menor* que —1.
LOG(X): da el logaritmo (en base e) de X.
RND(X): da un número aleatorio comprendido entre el 0 y el 1. Los efectos del valor de X varían bastante de un BASIC a otro. En el capítulo siguiente se analiza más a fondo esta función.
SIN(X): da el seno de X. (Véanse las notas relativas a la función COS).
SQR(X): da la raíz cuadrada positiva de X. El valor de X ha de ser positivo.
TAB(X): es una función muy especial porque, en vez de dar un valor, hace que el siguiente carácter se imprima en la columna X a no ser que en ese momento haya sido sobrepasada esa posición. La mayoría de las versiones de BASIC solo permiten usar la función TAB en instrucciones PRINT.
TAN(X): da la tangente de X. En el caso de que ese microordenador no tuviera la función TAN recuérdese que tan (X) = sen (X)/cos (X).

Hay, por supuesto, muchas más funciones no enumeradas aquí (todas las funciones de cadena por ejemplo) pero cada uno puede ahora buscarlas todas en el manual de BASIC de su microordenador.

Para que sirva de ejemplo de funciones puestas por el usuario, el siguiente programa traza curvas senoidales (tema tratado con más extensión en el capítulo siguiente).

```
10 X = 0
20 N = 20
30 PRINT TAB(INT((SIN(X) + 1)*N));"*"
40 X = X + 0.5
50 GOTO 30
```

Aunque se trata de un programa muy corto merece la pena estudiarlo. La línea 30 es la más complicada, con tres funciones: SIN(X) da un valor comprendido entre más y menos uno. Sumando una unidad y multiplicándolo por N se convierte en un número comprendido entre 0 y 2*N. La función INT lo convierte en un número completo y la función TAB hace que salga impreso el signo de multiplicar en la columna apropiada. Este método puede servir para trazar cualquier gráfica o función con valores comprendidos entre 0 y 2*N o que se pueden pasar a este campo.

Funciones definidas por el usuario

Supóngase que se quieren sacar las raíces cuadradas de una ecuación de segundo grado. Considérese por ejemplo:

$$ax^2 + bx + c = 0$$

Si se recuerda la fórmula que brinda la solución, es fácil de resolver. La primera raíz viene dada en BASIC por

```
10 R1 = (—B + SQR(B*B — 4*A*C))/(2*A)
```

y la segunda raíz viene dada por

```
20 R2 = (—B — SQR(B*B — 4*A*C))/(2*A)
```

Todo sería mucho más fácil si hubiera una función BASIC llamada ROOT que diera la primera raíz de una ecuación de segundo grado. Por ejemplo,

```
10 R1 = ROOT(A,B,C)
```

Por desgracia, el BASIC no llega tan lejos pero sí permite que el usuario defina una función que haga lo mismo. (Yo solo conozco una versión BASIC que no permite al usuario definir funciones, y es el BASIC del ZX81. Sin embargo, algunas versiones tienen restricciones en cuanto al tipo de funciones que es posible definir. El BASIC del DRAGON o del Tandy Color, por ejemplo, solo permite trabajar con un parámetro). Para definir una función se utiliza la instrucción DEF FN, así:

DEF FNx (lista de parámetros) = definición de la función

La x puede ser sustituida por un nombre de variable válido, en nuestro caso de una sola letra. La lista de parámetros es simplemente una lista de variables separadas por comas y la definición de la función es cualquier expresión aritmética que entre en una línea. Por ejemplo, si se declara la función FNR como la raíz primera de una ecuación de segundo grado, así:

```
10 DEF FNR(A,B,C) = (—B + SQR(B*B — 4*A*C))/(2*A)
```

entonces se podrá obtener la primera raíz de cualquier ecuación de segundo grado añadiendo la línea

```
20 PRINT FNR(A,B,C)
```

Es importante advertir que las letras o nombres utilizados en la lista de parámetros no tienen nada que ver con las variables del mismo nombre utilizadas en cualquier otra parte del programa. Una vez definida, una función aportada por el usuario se puede utilizar exactamente igual que una función incorporada. Concretamente, se puede utilizar cualquier variable como parámetros (no solo los enumerados en la lista de parámetros) y la función se puede utilizar tantas veces como se quiera. Por ejemplo, suponiendo que ha sido hecha ya la declaración de la función FNR definida anteriormente en el programa, todas las líneas siguientes, son correctas:

```
10 A = FNR(A,B,C)
20 A = FNR(P,C,D)
30 PRINT FNR(X,Y,Z)*FNR(T,R,F) + 3
```

He aquí otro punto relativo al empleo de parámetros en funciones: las variables utilizadas en la definición de la función, que no están nombradas en la lista de parámetros, *son consideradas* las mismas variables en el programa principal. Por ejemplo,

```
10 DEF FNA(N) = N*V
20 V = 2
30 PRINT FNA(3)
```

sacará impreso un 6. El parámetro N es puesto a 3 al emplear la función en la línea 30 pero la variable V es puesta a 2 por la línea 20 y no resulta modificada por el uso de la función.

A modo de indicación de la forma de utilizar las funciones de una forma más imaginativa considérese el problema de definir una función que permita hallar las dos raíces de una ecuación de segundo grado. Puede parecer imposible pues una función sólo puede dar una respuesta cada vez pero véanse las siguientes líneas de programa:

```
10 DEF FNR(A,B,C,N) = (—B + N*SQR(B*B — 4*A*C))/
(2*A)
20 PRINT "TECLEAR LOS TRES COEFICIENTES";
30 INPUT A,B,C
40 PRINT "LA PRIMERA RAIZ ES ";FNR(A,B,C,1)
50 PRINT "LA SEGUNDA RAIZ ES ";FNR(A,B,C,—1)
```

El truco consiste en incluir un parámetro adicional para indicar qué raíz ha de ser calculada, por ejemplo el parámetro N. Si N es 1 se calcula la primera raíz y si N es menos uno se calcula la segunda raíz.

Este ejemplo sirve también para poner de manifiesto la principal limitación de las funciones del BASIC. Siguiendo con el ejemplo anterior, cabe la posibilidad de que no haya ninguna raíz real de una ecuación de segundo grado. Esta situación sale a la luz si la expresión interna a la función SQR proporciona un número negativo. Es de todos conocido que la función SQR no puede operar con números negativos por lo que tal situación hará que el programa

genere un mensaje de error y se pare. La solución a este problema estaría en incluir un test (por ejemplo, una instrucción IF) que salte la función SQR en caso de problemas pero, por supuesto, esto es imposible porque las definiciones de funciones solo pueden incluir una línea de BASIC.

Las subrutinas

En BASIC las subrutinas no están tan desarrolladas ni son de tanta utilidad como las prestaciones ofrecidas en otros lenguajes, pero, de todas formas, vale la pena conocerlas. Una subrutina es un trozo del programa que realiza una tarea que se necesita hacer con frecuencia. Por ejemplo, si durante un programa es necesario poner a cero una matriz muchas veces, entonces en vez de utilizar

```
10 FOR I = 1 TO N
20 A(I) = 0
30 NEXT I
```

cada vez, sería mejor usar una sola línea que llamar a ese fragmento del programa. Se podría hacer escribiendo ese fragmento del programa comenzando con un número de línea alto, por ejemplo el 1000, y saltando a él mediante una GOTO cada vez que fuera necesario. El problema de este método es la vuelta al punto del programa que sigue a la GOTO. ¡El que no lo crea que haga la prueba! Para facilitar las cosas el BASIC tiene dos instrucciones más:

```
GOSUB número de línea
```

y

```
RETURN
```

La GOSUB opera como una GOTO en el sentido de que pasa el control al número de línea indicado pero además guarda para uso posterior el número de la línea que sigue a la GOSUB. La instruc-

ción RETURN averigua el número de línea guardado por la última GOSUB y a él transfiere el control. Un ejemplo lo dejará bien claro:

```
10 DIM A(20)
20 GOSUB 1000
30 PRINT "PUESTA A CERO DE LA MATRIZ"
40 A(3) = 3
50 GOSUB 1000
60 PRINT "NUEVA PUESTA A CERO DE LA MATRIZ"
70 STOP
1000 FOR I = 1 TO 20
1010A(I) = 0
1020 NEXT I
1030 RETURN
```

En la línea 20 el control pasa a la línea 1000 y la matriz es puesta a cero. La instrucción RETURN de la línea 1030 de la subrutina hace que el control pase a la línea 30, la línea siguiente a la GOSUB más reciente. La línea 50 hace otra llamada a la subrutina y la RETURN devuelve el control a la línea 60.

Otros puntos a destacar sobre las subrutinas son:

(a) Una subrutina puede llamar a otra subrutina y la instrucción RETURN se cuidará de transferir el control al sitio adecuado al final de cada una.

(b) Una RETURN sin la presencia de una GOSUB anterior será causa de un error.

(c) Todas las variables incorporadas a una subrutina son las mismas variables utilizadas en el resto del programa (incluyendo otras subrutinas).

El empleo frecuente de subrutinas para substituir a fragmentos de programa no es la única forma de empleo de las subrutinas. De hecho el empleo de las subrutinas ha sido desarrollado hasta convertirse en una filosofía, o incluso una ciencia del método de programación conocido como «Programación estructurada» (Top Down Structured Programming), una de cuyas principales ideas es el empleo casi obsesivo de subrutinas a la mínima oportunidad. El resultado es que estos programas están constituidos principalmente por

una larga lista de llamadas a subrutinas que facilitan la comprensión del programa y su modificación en el futuro. Si se quieren escribir buenos programas la regla a seguir es: ¡usar una subrutina a no ser que haya una buena razón para no hacerlo!

En los programas largos que irán saliendo se encontrarán buenos ejemplos de este tipo de programación.

Las expresiones

Uno de los primeros temas analizados en este libro era la forma de efectuar operaciones aritméticas en BASIC, extendido después a las expresiones aritméticas que incluían variables y ahora a las funciones. La verdad es que cuando en BASIC se cambia cualquier información, el agente del cambio es una expresión. Por ejemplo, en el capítulo anterior, que analizaba el empleo de las cadenas, se usaban la concatenación de cadenas y las funciones de cadenas componiendo expresiones de cadenas. Más tarde se describirá que hay otra forma de expresiones en BASIC: las expresiones lógicas. En resumen, una expresión es una especie de receta para combinar y alterar datos de cualquier tipo. Si se da el caso de que los datos son números, se está ante expresiones aritméticas; si los datos son caracteres se está ante expresiones de cadenas y, finalmente, si los datos son valores lógicos se está ante una expresión lógica. Hay que destacar que una expresión contiene siempre la idea de hacer algo. He ahí la principal diferencia entre una expresión aritmética y una fórmula algebraica. La fórmula puede representar alguna verdad (que una cantidad es igual a otra o que dos cantidades están relacionadas) pero una expresión aritmética es algo que el microordenador está obligado a calcular.

La idea de efectuar una expresión, o evaluarla, merece un análisis más detallado. Si el microordenador efectúa una expresión aritmética, por ejemplo, el resultado es un número; y ¿por qué tratar este número de una forma distinta que cualquier otro número que se haya usado en un programa BASIC? Cabe preguntarse por qué, por ejemplo, no se deben escribir cosas como

```
PRINT 2*SQR(3) + 5
IF A + 34>E/2 THEN...
FOR I = 55 + J TO N —1 STEP SIN(.5)
```

La respuesta es que el BASIC, en la mayoría de las versiones, lo permite. La regla general en BASIC es: donde se pueda utilizar un número, se puede utilizar una expresión aritmética y viceversa. De forma similar, donde se pueda utilizar una cadena, se puede emplear una expresión de cadenas. Y, por último, como se verá más adelante, donde se pueda utilizar un valor lógico, se puede utilizar una expresión lógica. Hasta aquí estas libertades han sido presentadas como casos especiales. Cuando se presentó el bucle FOR se dijo explícitamente que se podían utilizar expresiones aritméticas para los valores de comienzo y de fin y para los intermedios los de cada paso o etapa. Sin embargo se trataba simplemente de un caso concreto de aplicación de la regla general reseñada anteriormente. Algunas versiones BASIC llevan este principio a su conclusión lógica y permiten escribir expresiones como GOTO X*3 + 34, pero esto es algo que no conviene estimular porque hace los programas más difíciles de depurar.

Preguntas de autoevaluación

1. Escríbase un programa que imprima una tabla de raíces cuadradas positivas comprendidas entre dos valores dados y con una secuencia dada.

2. Modifíquese el programa de la pregunta **1** con vistas a que imprima una tabla de los valores seno y coseno.

3. Si la versión de BASIC que se utiliza tiene funciones definidas por el usuario, escríbase una función que saque el logaritmo en base 10 de un número partiendo de que

$$\log_{10} x = \log_e x \log_{10} e$$

dado que $\log_{10} e = 0.434295$

4. Escríbase un programa que efectúe

S = x — x↑3/6 + x↑5/120 — x↑7/5040

para un valor dado de x. Utilizar o una subrutina o una función definida por el usuario para calcular el resultado. Si se quiere, se puede comparar el resultado obtenido con SIN(x), donde x está en radianes.

6
PRACTICAS CON EL BASIC

A estas alturas conocemos ya suficiente BASIC para abordar proyectos de mayor entidad. No obstante, la electrónica es una materia que implica el empleo de algunas técnicas —los números complejos por ejemplo— con las que nunca se había pensado que tuviera que enfrentarse el BASIC. En este capítulo se analizarán algunas de las ideas que caracterizan el empleo del BASIC en electrónica y se hará mediante ejemplos sencillos.

Trazado de curvas

En el anterior capítulo se presentó y analizó un programa que trazaba una onda senoidal continua sirviéndose de la función TAB. Aunque muchos microordenadores ofrecen buenas prestaciones gráficas, este tipo de trazado primario suele ser muchas veces la mejor vía. Hay muchas razones para ello. La total ausencia de estandarización de las instrucciones de gráficos hace que la escritura de programas que emplean esas instrucciones de gráficos sea muy específica para cada microordenador. Además, disponer de una impresora gráfica sigue siendo la excepción y no la regla, y si una gráfica no puede ser impresa, su utilidad es muy limitada. Y por último, el tipo de gráficas que se utilizan en electrónica no siempre necesitan ser dibujadas con la alta precisión de los sistemas gráficos

modernos. Muchas veces lo que interesa es únicamente la forma general y los valores de unos pocos puntos críticos.

Es bien sabido que una de las ideas centrales de la electrónica es que se puede generar cualquier forma de onda simplemente sumando ondas senoidales (y cosenoidales). Aunque se trata de una teoría bien conocida siempre es mejor ver la teoría en acción. En tal sentido, el siguiente programa traza la suma de una serie de ondas senoidales.

```
  10 LET K=0
  20 LET L=40
  30 DIM Y(50)
  40 INPUT A
  50 INPUT B
  60 GO SUB 1000
  70 GO SUB 2000
  80 GO SUB 3000
  90 LET K=K+1
 100 GO TO 40
1000 LET X=1
1010 FOR T=0 TO 2*3.14159 STEP 2
*3.14159/50
1020 LET Y(X)=Y(X)+A*SIN (K*T)+B
*COS (K*T)
1030 LET X=X+1
1040 NEXT T
1050 RETURN
2000 LET M=Y(1)
2005 LET S=Y(1)
2010 FOR I=1 TO 50
2020 IF Y(I)>M THEN LET M=Y(I)
2030 IF Y(I)<S THEN LET S=Y(I)
2040 NEXT I
2050 IF M=S THEN LET S=M/2+1
2060 RETURN
3000 FOR I=1 TO 50
3010 PRINT TAB ((Y(I)-S)*L/(M-S)
+.5);"*"
3020 NEXT I
3030 RETURN
```

Este programa se ha escrito utilizando subrutinas que desarrollan las tareas principales. Conviene darse cuenta de cómo el empleo de las subrutinas hace que el programa sea mucho más fácil de entender. El programa principal, líneas 10 a 100, está constituido casi únicamente por una lista de llamadas a subrutinas. La subrutina 1000 tiene la función de sumar el siguiente término seno y coseno de la serie. La variable K sirve para llevar la cuenta del número de términos, y A y B son las amplitudes especificadas por el usuario para

el seno y para el coseno respectivamente. La subrutina 2000 halla los valores máximo y mínimo contenidos en la matriz Y. Finalmente, la subrutina 3000 traza la gráfica usando la función TAB tal como se ha descrito en el capítulo anterior. La única diferencia estriba en que esta rutina de trazado de gráficas, funcionará sea cual sea la lista de valores almacenados en Y, pues la gráfica cambia automáticamente de escala para que entre en la pantalla. Lo hace sirviéndose de los valores mayor y menor almacenados en M y S y del número de caracteres de una línea de la pantalla, almacenado en L.

El programa principal está escrito en forma de bucle. Cada pasada por el bucle pide al usuario los valores de A y B y luego suma un término de la forma

A*SIN(K*T) + B*COS(K*T)

a los valores actuales de Y. De esta manera se puede ir viendo la forma que va tomando la serie según va incorporando más y más términos de frecuencia cada vez más alta (adviértase que K aumenta en una unidad a cada pasada por el bucle).

A modo de demostración del programa en acción, hágase la prueba introduciendo los valores siguientes para A y B:

K	A	B
0	0	0
1	1	0
2	0	0
3	0.333	0
4	0	0
5	0.2	0
6	0	0
7	0.143	0

Se puede ver en la pantalla una aproximación cada vez más clara a una onda cuadrada. Si se quiere llevar la serie a más términos, la regla a seguir es: los términos pares tienen valores cero para A y B y los términos impares tienen los valores A = 1/K y B = 0. Utilizando únicamente métodos BASIC muy sencillos se ha hecho un programa que muestra en acción las series de Fourier.

Los histogramas

Otra forma de presentación gráfica de suma utilidad es el histograma, incluso más fácil de generar que una gráfica senoidal. Supongamos que se quiere generar un histograma que represente los datos 6, 10 y 3. Lo único que hay que hacer es sacar impresa una línea de 6 asteriscos, después una línea de 10 y luego una de 3. Es decir,

```
I******
I**********
I***
```

Probablemente ya se habrá adivinado que un programa que haga esto contará con dos bucles FOR, uno para imprimir el número necesario de líneas y el otro para imprimir el número de asteriscos de cada línea. Si los datos del ejemplo anterior están almacenados en la matriz H, el programa siguiente generará el histograma:

```
1000 FOR I = 1 TO 3
1010 PRINT "I";
1020 FOR J = 1 TO H(I)
1030 PRINT "*";
1040 NEXT J
1050 PRINT
1060 NEXT I
```

El bucle FOR exterior, que comienza en la línea 1000, imprimirá tres líneas y el bucle FOR interior, que comienza en la línea 1020, imprimirá H(I) asteriscos. El único punto que merece la pena mencionar es la función del punto y coma de las líneas 1010 y 1030. Si se pone fin a una instrucción PRINT con un punto y coma, las instrucciones PRINT que vengan a continuación siguen imprimiendo donde la primera lo dejó. En otras palabras, el punto y coma suprime el comienzo normal en otra línea tras una instrucción PRINT.

Aunque este programa constituye el método básico de trazado de histogramas es, sin embargo, de muy escasa utilidad. Concreta-

mente, no puede enfrentarse a una amplia gama de datos sin sobrepasar las dimensiones de la pantalla. La respuesta a este problema está en explorar la matriz H buscando el valor más grande y en utilizar este valor para hacer el histograma a la escala adecuada para que entre en la pantalla. Esta puesta a escala es bastante sencilla: si el valor más grande es M (de Mayor) y el número de asteriscos de la línea más larga es L, el número de asteriscos que han de imprimirse para H(I) es

INT(H(I)*L/M)

donde INT toma la parte entera del resultado para evitar cualquier intento de imprimir un número fraccionario de asteriscos. El programa es el siguiente:

```
  10 DIM H(50)
  20 LET L=40
  30 PRINT "CUANTOS VALORES ";
  40 INPUT N
  50 FOR I=1 TO N
  60 PRINT "VALOR ";I;" = ";
  70 INPUT H(I)
  80 NEXT I
  90 GO SUB 1000
 100 GO TO 30
1000 GO SUB 2000
1010 FOR I=1 TO N
1020 PRINT "I";
1030 LET A=INT (H(I)*L/M)
1040 IF A=0 THEN GO TO 1080
1050 FOR J=1 TO A
1060 PRINT "*";
1070 NEXT J
1080 PRINT
1090 NEXT I
1100 RETURN
2000 LET M=H(1)
2010 FOR I=1 TO N
2020 IF M<H(I) THEN LET M=H(I)
2030 NEXT I
2040 RETURN
```

El programa principal se limita a leer los datos en H y llamar después a la subrutina 1000 para que trace el histograma. La subrutina 1000 es básicamente la anterior con la sola diferencia de que ahora llama a la subrutina 2000 para averiguar el valor máximo de H y se sirve de este resultado para poner a escala el histograma tal como

ha quedado descrito. El único punto adicional de interés es el empleo de la instrucción IF de la línea 1040 para saltar el bucle FOR interior cuando no hay asteriscos a imprimir.

Los números aleatorios

La función de BASIC RND salió en el capítulo anterior pero sin que se dieran muchas explicaciones sobre la forma de usarla. El problema principal que plantea el empleo de la instrucción RND es precisamente cómo escribirla. Algunas versiones de BASIC usan la forma RND, otras usan la forma RND(0) y hay otras que usan la forma RND(—1), todas con el mismo significado. La función RND generará números aleatorios comprendidos entre el 0 y el 1 pero no incluyendo este último. Cada uno ha de averiguar la forma exacta de la función que utiliza su BASIC. Una vez averiguada, hágase la prueba siguiente:

```
10 PRINT RND
20 GOTO 10
```

y compruébese que se obtienen números aleatorios comprendidos entre el 0 y el 1.

Como no es normal que se necesiten números aleatorios comprendidos entre el 0 y el 1, vale la pena conocer la forma de pasar de esta gama a otra de más utilidad. En caso de necesitar números aleatorios enteros, comprendidos entre A y B, utilícese la forma

```
INT(RND*(B — A + 1)) + A
```

En caso de necesitar números fraccionarios, comprendidos entre A y B, utilícese la forma

```
RND*(B — A) + A
```

La aplicación principal de los números aleatorios en electrónica es la generación de una gama representativa de datos de prueba. Por ejemplo, en vez de teclear valores para llenar la matriz H a fin

de comprobar el programa de trazado dado anteriormente, cabría la solución de generar números aleatorios, cambiando las líneas 60 y 70 por

```
60 H(I) = INT(RND*100)
70 PRINT "VALOR";I;" = ";H(I)
```

que llenan la matriz con números comprendidos entre el 0 y el 99.

La principal característica de los números generados por una RND radica (por lo menos en teoría) en que todos tienen las mismas posibilidades de salir. Es muy frecuente el caso de querer comprobar la influencia que tendrán en el funcionamiento de un circuito ligeras variaciones con respecto al valor ideal de un componente, generando a tal fin números de la forma «valor ideal + error», donde «error» es el factor aleatorio que modifica el valor ideal. Se da casi siempre el caso de que el término error tiene una distribución normal en vez de la distribución uniforme de números generados por una RND. Se pueden generar números con una distribución normal de desviación estándar S mediante

```
SQR(2)*(RND + RND + RND + RND + RND + RND—3)*S
```

Es solo una aproximación pero es una aproximación suficientemente buena para la mayoría de los casos.

Aritmética compleja

Muy pocas versiones de BASIC ofrecen medios para el tratamiento de números complejos. Sin embargo, si se tratan la parte real y la imaginaria separadamente, es muy fácil crear subrutinas, o funciones definidas por el usuario, para la adición, la sustracción, la multiplicación y la división complejas.

Si la versión de BASIC con la que se esté operando permite poner nombres de dos letras a las variables, una buena solución consiste en crear dos variables para cada variable compleja. La primera tendría un nombre que acabara en R y correspondería a la parte real y la segunda tendría un nombre que acabara en I y corres-

pondería a la parte imaginaria. Por ejemplo, FR y FI podrían ser los nombres de las variables que guardaran las partes real e imaginaria de una frecuencia compleja.

La adición y la sustracción complejas son fáciles. Si w y v son números complejos,

$$re(w + v) = re(w) + re(v)$$
$$im(w + v) = im(w) + im(v)$$

y

$$re(w - v) = re(w) - re(v)$$
$$im(w - v) = im(w) - im(v)$$

donde re es la parte real e im la parte imaginaria. En otras palabras, para sumar dos números complejos lo único que hay que hacer es sumar las partes reales e imaginarias separadamente y, de un modo similar, en la sustracción se opera con las partes real e imaginaria separadamente. Es tan sencillo que no hay necesidad de escribir subrutinas o funciones especiales para la adición y la sustracción. No obstante, cuando le llegue el turno a la multiplicación y a la división las cosas cambian. Si v y w son números complejos,

$$re(v*w) = re(v)*re(w) - im(v)*im(w)$$
$$im(v*w) = re(v)*im(w) + im(v)*re(w)$$

y

$$re(v/w) = \frac{re(v)*re(w) + im(v)*im(w)}{re(w)*re(w) + im(w)*im(w)}$$

$$im(v/w) = \frac{im(v)*re(w) - re(v)*im(w)}{re(w)*re(w) - im(w)*im(w)}$$

Si la versión de BASIC con la que se está trabajando soporta funciones definidas por el usuario, con cuatro parámetros, se puede usar la forma

```
DEF FNM(VR,VI,WR,WI) = VR*WI — VI*WI
```

y

```
DEF FNN(VR,WR,VI,WI) = VR*WI + VI*WR
```

para las partes real e imaginaria de la multiplicación VR,VI y WR,WI, respectivamente, y

```
DEF FND(VR,VI,WR,WI) = (VR*RW + VI*WI)/(WR*WR
+ WI*WI)
```

y

```
DEF FNE(VR,VI,WR,WI) = (VI*WR — VR*WI)/(WR*WR
+ WI*WI)
```

para las partes real e imaginaria de la división de VR,VI por WR,WI, respectivamente. Si no es posible usar funciones definidas por el usuario de cuatro parámetros, se puede elegir entre tener que escribir las expresiones cuando sea necesario o usar subrutinas.

Dos operaciones estándar con números complejos que surgen frecuentemente en electrónica son la conversión de la forma cartesiana a la polar y la contraria (ver figura 6.1). Si v es un número complejo en forma cartesiana, entonces

```
r = SQR(re(v)↑2 + im(v)↑2)
t = ARCTAN(im(v)/re(v))
```

donde r es el módulo y t es el argumento (o ángulo de fase). La mayoría de las versiones de BASIC incluyen la función ARCTAN (arcotangente) pero algunas veces con el nombre ATAN o incluso ATN. Hay que comprobarlo en cada versión BASIC. La conversión de formas polares a cartesianas es más fácil todavía:

```
re(v) = r*COS(t)
```

y

```
im(v) = r*SIN(t)
```

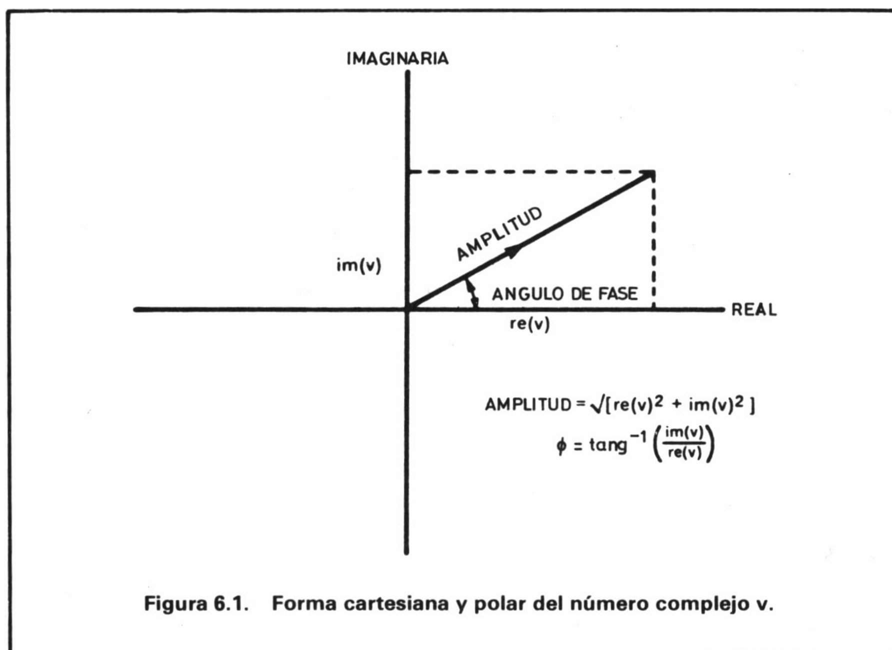


Figura 6.1. Forma cartesiana y polar del número complejo v.

Considérese, a modo de ilustración de algunos de estos métodos, el problema del cálculo de la impedancia del circuito LCR mostrado en la figura 6.2. Después de las explicaciones anteriores se está en disposición de utilizar las ecuaciones y derivar la magnitud de la impedancia y el ángulo de fase pero es más fácil, si se tiene un microordenador, operar con impedancias complejas. El empleo de los números complejos simplifica considerablemente la teoría de los circuitos que llevan condensadores e inductancias. La idea es, que si se está dispuesto a trabajar con impedancias complejas, se pueden calcular las impedancias totales, utilizando las mismas fórmulas que se usarían para las resistencias puras. Luego la impedancia del circuito LCR serie de la figura 6.2 es sencillamente

$$Z = R + ZL + ZC$$

donde ZL y ZC son las impedancias de la inductancia y del condensador respectivamente. Utilizando las fórmulas $ZL = j\omega L$ y $ZC = -j/\omega C)$ sale

$$Z = R + j(\omega L - 1/(\omega C))$$

donde ω es la frecuencia angular. Ahora, para investigar la respuesta del circuito LCR lo único que hay que hacer es efectuar esta ecuación para diversos valores de ω. Pero para que los resultados tengan sentido, es necesario convertir las impedancias complejas a la forma polar y luego trazar dos gráficas, una de la impedancia y la otra del ángulo de fase. El programa siguiente realiza estos procesos.

```
  10 DIM ZR(50),ZI(50)
  20 LET P=40
  30 LET L=.001
  40 LET C=.000001
  50 LET M=100
  60 INPUT R
  70 GO SUB 1000
  80 GO SUB 2000
  90 GO SUB 3000
 100 GO TO 20
1000 LET K=1
1010 FOR F=1000 TO 20000 STEP 50
0
1020 LET ZR(K)=R
1030 LET ZI(K)=2*3.14159*F*L-1/(
2*3.14159*F*C)
1040 LET K=K+1
1050 NEXT F
1060 RETURN
2000 FOR I=1 TO 25
2010 LET A=SQR (ZR(I)*ZR(I)+ZI(I
)*ZI(I))
2020 PRINT TAB (A*P/M);"*"
2030 NEXT I
```

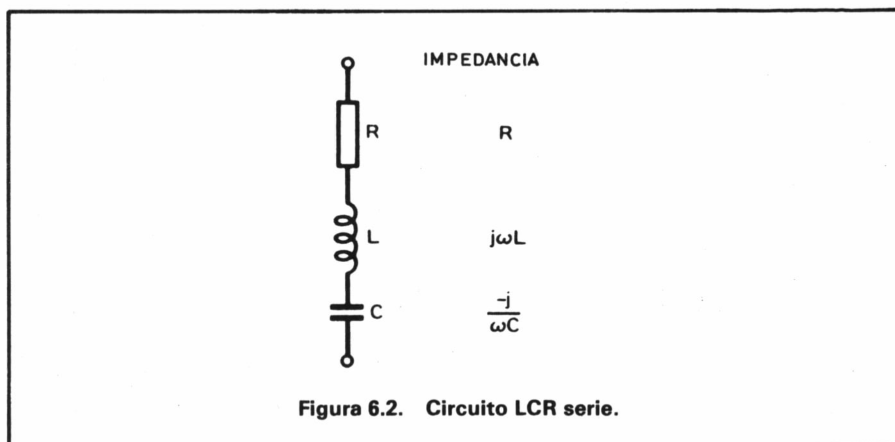


Figura 6.2. Circuito LCR serie.

```
2040 RETURN
3000 FOR I=1 TO 25
3010 LET A=ATN (ZI(I)/ZR(I))+2*3
.14159
3020 PRINT TAB (A*P/4/3.14159);"
X"
3030 NEXT I
3040 RETURN
```

La subrutina 1000 calcula la impedancia compleja para una amplia gama de frecuencias y guarda la parte real en ZR y la parte imaginaria en ZI. (Si la versión de BASIC con la que se está trabajando no soporta nombres de dos letras, lo único que hay que hacer es dar otros nombres a las variables). Las subrutinas 2000 y 3000 trazan las gráficas de la magnitud y del ángulo de fase de la impedancia mediante técnicas que ya han sido descritas a fondo en capítulos anteriores. El programa tal como está, sólo permite al usuario alterar el valor de R con vistas a analizar su efecto de amortiguación en la resonancia. La resonancia propiamente dicha se ve claramente como una depresión en la gráfica de la magnitud y como un cambio de fase en la gráfica del ángulo de fase.

Se puede utilizar este programa para experimentar con los efectos que produce el cambio de los valores de L y C, pero hay que tener en cuenta que la escala de las gráficas es elegida en la línea 50 y que es casi seguro que será necesario cambiarla para mantener la gráfica dentro de los márgenes de la pantalla. Incluso quienes estén familiarizados con las matemáticas de los circuitos LCR serie, pueden aprender muchas cosas «jugando» con este programa. Por ejemplo, si la frecuencia de resonancia está determinada por el producto de L y C, ¿qué efecto tendrá en la respuesta mantener el producto invariable y cambiar las proporciones de L y C?

Se puede investigar la respuesta de similares combinaciones de L, C y R utilizando este programa. Lo único que hay que recordar es hacer el cálculo de la impedancia total, como si todos los componentes fueran resistencias puras y después sustituir $j\omega L$ y $-j/\omega C$ por la impedancia inductiva y la impedancia capacitiva respectivamente. Con un microordenador los circuitos de corriente alterna pasivos son fáciles de entender.

Las prácticas de programación

Este capítulo ha presentado varios programas sencillos tanto para ilustrar la programación como con la finalidad de sugerir el tipo de actitud que hay que adoptar ante el microordenador. Ninguno de los programas pueden ser considerados completos en ningún sentido y es el lector el que, aceptando el reto, debe convertirlos en productos acabados y, por lo tanto, útiles. Se aprende mucho leyendo y perfeccionando los programas hechos por otros y es importante acostumbrarse a ello desde el principio. Si no se hace así muchas veces no se tendrá más remedio que escribir programas que ya existen simplemente porque no hacen exactamente lo que se quiere que hagan.

El microordenador es una herramienta poderosa que permite explorar los cálculos que hay detrás de la electrónica. Y componiendo una colección de subrutinas y técnicas del tipo de las presentadas en este capítulo se puede incluso reducir el trabajo de escribir programas incorporándolas donde sean necesarias.

7

LA UTILIZACION DEL BASIC EN LA ELECTRONICA DIGITAL

Son muchos los que creen que los programadores y los microordenadores se pasan casi todo el tiempo operando con lógica. Los que hayan leído los anteriores capítulos saben ya que eso no es ni mucho menos cierto. Para escribir programas en BASIC, o en cualquier otro lenguaje informático, no es necesario ningún tipo de conocimiento de lógica tradicional. Pero para el tema de la electrónica digital las cosas son muy distintas. El diseño de cualquier equipo digital exige, por lo menos, un conocimiento intuitivo de lógica. El BASIC puede servir de ayuda para enfrentarse con la electrónica digital porque puede evaluar expresiones lógicas de la misma forma que puede evaluar expresiones aritméticas. Una vez que se conozca la lógica en BASIC será fácil encontrar muchas otras aplicaciones, aparte del diseño de electrónica digital, porque si es cierto que la programación sirve de ayuda en el campo de la electrónica, también es cierto que la práctica de la lógica sirve de ayuda en el campo de la programación.

La lógica Booleana

Hay muchos tipos diferentes de lógica pero la que nos ocupa aquí y ahora es una de las más sencillas: la lógica Booleana. La

lógica de Boole está basada en el simple hecho de que una sentencia es cierta o es falsa. Para que la lógica Booleana funcione, es preciso que en todos estos puntos sólo haya dos posibilidades: cierto o falso. Considérese la sentencia de BASIC A > 0. ¿Quiere esto decir que la variable A es mayor que cero? Supongamos que A ha sido puesta a uno en una parte anterior del programa. Entonces A > 0 es cierta. Pero si se hace A igual a menos uno, entonces A > es falsa. Ha de quedar bien claro ya que A > 0 no significa que se esté afirmando que A es mayor que 0. Se trata de una simple sentencia que puede ser cierta o falsa. En este sentido, A > 0 no es de hecho una sentencia o una instrucción sino una expresión a evaluar (ver Capítulo 5). Una sentencia que puede ser cierta o falsa recibe el nombre de expresión Booleana. Se puede utilizar, por ejemplo, una expresión como A > 0 en una línea como ésta:

```
10 IF A > 0 THEN PRINT "A ES MAYOR QUE CERO"
```

En este libro han salido ya expresiones Booleanas dentro de instrucciones IF. La forma general de la instrucción IF se puede expresar ahora así:

IF expresión Booleana es cierta, THEN sentencia

Las expresiones Booleanas que conocemos ya están todas basadas en los operadores relacionales presentados cuando se trataron las instrucciones IF. Son:

= igual a	A = B
< menor que	A < B
> mayor que	A > B
< > no igual que	A < > B
< = menor o igual que	A < = B
> = mayor o igual que	A > = B

Los nuevos conocimientos adquiridos ahora en este tema dan entrada a la nueva definición de estos puntos como sentencias que pueden ser ciertas o falsas.

Hay que destacar que el signo igual tiene dos significados en BASIC. Se puede utilizar con el significado de asignación, es decir LET A = 0 ó puede ser empleado como operador relacional, es decir, IF A = 0 THEN etc. En el primer caso, A es puesto a cero y en el segundo, A no sufre ninguna modificación y únicamente sale como resultado de un cierto o un falso. Hay que recordar también, que los operadores relacionales se pueden utilizar para comparar cadenas, igual que para comparar números. Véase en tal sentido el Capítulo 4.

Los operadores Booleanos

Hasta este momento las expresiones Booleanas no nos han dado nada en concreto, únicamente la posibilidad de hacer una re-interpretación de hechos ya conocidos. Si ésta fuera la única razón que justificara el empleo de la lógica no merecería la pena molestarse en estudiarla. Pero considérese el siguiente programa:

```
10 PRINT "INTRODUCIR DOS VALORES DE
   RESISTENCIAS";
20 INPUT R
30 INPUT S
40 IF R < 0 THEN PRINT "LA RESISTENCIA HA DE SER
   MAYOR QUE CERO"
50 IF S < 0 THEN PRINT "LA RESISTENCIA HA DE SER
   MAYOR QUE CERO"

(resto del programa)
```

En las líneas 40 y 50 se comprueba si los dos valores de las resistencias son menores que cero. Examinando estas dos líneas detenidamente se podría llegar a la conclusión de que hay alguna repetición porque pasa lo mismo en el caso de que cualquiera de las resistencias sea menor que cero, y, además, si las dos son menores que cero el mensaje sale dos veces. El lector puede, a modo de ejercicio, intentar modificar el programa de manera que saque impreso el mensaje una sola vez, sea el que sea.

Es de sentido común pensar que sería mejor poder escribir la línea siguiente en vez de las líneas 40 y 50:

```
40 IF R < 0 OR S < 0 THEN PRINT "LA RESISTENCIA
   DEBE SER MAYOR QUE CERO"
```

De hecho la parte de la instrucción comprendida entre IF y THEN sigue siendo una expresión Booleana, es decir, puede ser cierta o falsa. Para comprobar que lo dicho es cierto lo mejor es analizar qué es lo que hace que la instrucción IF ejecute la instrucción que sigue a THEN.

R < 0	S < 0	THEN ejecución de la instrucción siguiente
Falsa	Falsa	NO
Falsa	Cierta	SI
Cierta	Falsa	SI
Cierta	Cierta	SI

Si se examina la tabla queda bien claro que la instrucción que sigue a THEN es ejecutada si cualquiera de las dos desigualdades es cierta y esto es lo que OR significa. Volviendo a la definición de la instrucción IF, la expresión Booleana R < 0 OR S < 0 debe ser cierta si ha sido ejecutada la instrucción que viene a continuación de THEN. Ahora se puede escribir la tabla anterior así:

R < 0	S < 0	R < 0 OR S < 0
Falsa	Falsa	Falsa
Falsa	Cierta	Cierta
Cierta	Falsa	Cierta
Cierta	Cierta	Cierta

Esto es lo que se llama una «tabla de verdad», una de las formas fundamentales de llegar a entender las expresiones lógicas. El ejemplo anterior, aparte de dar a conocer lo que es una tabla lógica, ha puesto al descubierto el primer operador lógico: OR.

Merece la pena detenerse un momento a pensar sobre las similitudes que hay entre lo que se acaba de hacer y la evaluación más familiar de las expresiones aritméticas. En aritmética se escribe A + B y todo el mundo entiende que hay que calcular el resultado de A + B para diversos valores de A y B. Se podría escribir una «tabla arit-

mética» enumerando todos los valores que pueden tener A y B y los resultados de hacer la operación A + B, pero se tardaría bastante tiempo porque los valores que pueden tomar A y B son infinitos. En el caso de las expresiones lógicas no hay ese problema pues cada variable sólo puede tomar uno de los dos valores (puede ser cierta o falsa), y, por lo tanto, se puede escribir la «tabla de verdad» correspondiente a A OR B con suma facilidad.

Las variables lógicas

Volviendo al tema de la analogía con la aritmética, hay que decir que los dos valores, cierto y falso, son constantes lógicas de la misma forma que 0, 1, 2, 3, 4, etc., son constantes aritméticas. Sería bueno que el BASIC diera la posibilidad de operar con variables lógicas que pudieran igualarse a constantes lógicas. Por ejemplo,

```
10 A = CIERTO
20 IF A THEN PRINT "A CIERTO"
```

Es posible que resulte difícil de leer y que parezca un poco extraño pero no debe haber problemas para saber de qué se trata. Se está utilizando la variable A como una variable lógica (que sólo puede almacenar los valores cierto/falso) y en la línea 10 es puesta a la constante cierto; en la línea 20 la instrucción IF evalúa la expresión Booleana A y encuentra que es cierta, luego es ejecutada la acción especificada por THEN.

El problema está en que la mayoría de los BASICs no tienen variables lógicas luego esta explicación no es del todo correcta. Es, sin embargo, correcta en la idea, tal como se verá más adelante. Aunque los dos valores que puede tomar una expresión Booleana suelen recibir el nombre de cierto y falso, los nombres son siempre totalmente arbitrarios. Se les podría llamar «plin» y «plon» o cualquier otra cosa sin que la lógica Booleana cambiara un ápice. Concretamente se les podría llamar 0 y 1 y utilizar las variables aritméticas normales para guardar valores lógicos. Este es el método que siguen la mayoría de los BASICs. La única complicación es que no todos están de acuerdo en usar el 0 y el 1, y algunos usan el 0 y el —1, ó el —1 y el +1, etc. La falta de normalización es irritante

pero no demasiado importante. En la tabla siguiente se puede ver un breve resumen del tratamiento que los diferentes BASICs dan a la lógica Booleana. (La columna «campo de evaluación» se explica más adelante).

BASIC	*Cierto*	*Falso*	*Campo de evaluación*
Microsoft V5	0	—1	—32768 a 32767
ZX81/Spectrum	1	0	Véase más adelante
Apple	1	0	No aplicable
BBC	—1	0	—65536 a 65535

En el resto de este libro se adoptará el 1 como cierto y el 0 como falso. Los cambios necesarios para hacer que los programas operen con otra convención son bastante fáciles y proporcionan una buena práctica para el tratamiento de la lógica. Con este punto decidido, el programa anterior se convierte en

```
10 A = 1
20 IF A THEN PRINT "A CIERTO"
```

Ahora ya es posible utilizar el ordenador para generar la tabla de verdad correspondiente a OR, así:

```
10 FOR I = 0 TO 1
20 FOR J = 0 TO 1
30 PRINT I;" ";J" " ; I OR J
40 NEXT J
50 NEXT I
```

Los operadores lógicos

Llegamos a nuestro primer operador lógico —OR— apelando al sentido común. Sería posible seguir aplicando el sentido común y sacar los demás operadores lógicos de uso más frecuente pero, para ahorrar tiempo y evitar el aburrimiento, he aquí un programa que saca impresas las tablas de verdad correspondientes a los operadores OR, AND y NOT:

```
10 PRINT "TABLAS DE VERDAD DE AND-OR-NOT"
20 FOR I = 0 TO 1
30 FOR J = 0 TO 1
40 PRINT I;" ";J;" ";I AND J;" ";I OR J;" ";NOT I
50 NEXT J
60 NEXT I
```

La palabra AND puesta entre dos variables lógicas hace que la expresión sea cierta solo si las dos variables son ciertas. NOT es un operador unitario (ver Capítulo 2) y simplemente cambia el valor de una variable por el otro valor, es decir, NOT cambia cierto por falso y falso por cierto. Del OR ya hemos tratado. Estos tres operadores lógicos básicos son todo lo que se necesita para expresar cualquier lógica Booleana. Dicho a un nivel más práctico AND, OR y NOT es lo único que hace falta para expresar cualquier conjunto de condiciones con una IF a fin de que sea ejecutada o no la sentencia que sigue a THEN.

Los operadores lógicos se pueden combinar en expresiones de una forma muy similar a los operadores aritméticos. (Lo único que puede plantear problemas es la prioridad de los operadores. La mayoría de los BASICs definen la siguiente escala de prioridades: NOT es el primero, después AND y por último OR, a no ser que haya paréntesis de por medio). Por ejemplo, NOR(I) AND I, NOT(A < 0) AND (B < 0), (A = 0) OR ((A = 1) AND (B = 0)) todas son expresiones lógicas válidas. Las reglas para la manipulación de estas expresiones no son difíciles de aprender. El que esté interesado en el tema podrá encontrar muchos libros sobre lógica sencilla que le servirán de ayuda para manipular lógica con facilidad. Para programación, sin embargo, no es necesario ser un experto en lógica: el microordenador puede encargarse de hacer todos los cálculos difíciles. Por ejemplo, si en un determinado momento se necesita saber la tabla de verdad correspondiente a la expresión NOT(I) AND(I), basta cambiar la línea 40 del programa anterior por la siguiente:

```
40 PRINT I;" ";J;" ";NOT(I) AND I
```

Si se tiene la suerte de poseer un ZX81 o un ZX Spectrum, o de utilizar una versión BASIC que cuente con la instrucción VAL o EVALuar, se puede cambiar el programa anterior por el siguiente:

```
10 PRINT A$
40 PRINT I;" ";J;" ";VAL(A$)
```

Este programa permite teclear cualquier expresión lógica con I y J e imprimir la tabla de verdad correspondiente, porque la función VAL calcula la expresión contenida en la cadena A$. Si se quiere calcular la tabla de verdad de una expresión de tres variables, lo único que hay que hacer es añadir un bucle FOR con una variable K, otro más para el caso de cuatro variables, otro más para cinco... hasta agotar la memoria o la paciencia.

La simplificación de expresiones

Hay una cosa que el microordenador no puede hacer, por lo menos fácilmente: es la simplificación de expresiones Booleanas. Algunas veces al calcular lo que debe contener una instrucción IF sale algo como esto:

```
10 IF NOT(A < 0) THEN ...
```

Uno podría haberse dicho a sí mismo «SI A no es menor que cero ENTONCES»... y la traducción de «no menor que cero» en BASIC es NOT(A < 0). No hay nada erróneo en este método de construcción de sentencias IF —todos lo hacemos así— pero el que haya hecho unos cuantos programas habrá advertido que NOT(A < 0) es lo mismo que A > = 0. Dicho en palabras, decir «A no es menor que cero» es lo mismo que decir «A es mayor o igual a cero». Haciendo este simple cambio se consigue que el programa sea un poco más sencillo y se ejecute con más rapidez.

Hay una multitud de simplificaciones de este tipo que se pueden aplicar a una expresión lógica compleja pero cada uno debe decidir si vale la pena hacerlo. Es algo muy similar a la simplificación de las expresiones aritméticas antes de incluirlas en un programa: no es erróneo poner A*B + A*C en vez de A*(B + C) pero la última forma es más rápida. Todos aprendimos a simplificar expresiones aritméticas en la escuela pero las expresiones lógicas son otra cosa. Afortunadamente nos salva el hecho de que las expresiones lógicas

que surgen en la práctica son bastante simples y se pueden modificar empleando el sentido común.

No obstante, puede ser útil conocer algunas reglas para la manipulación de expresiones Booleanas. Helas aquí:

Relaciones

1. NOT(A = B) = A< >B
2. NOT(A < B) = A> = B
3. NOT (A > B) = A = < B

Lógica

1. NOT(NOT(A)) = A
2. A OR (B AND C) = (A OR B) AND (A OR C)
3. A AND (B OR C) = (A AND B) OR (A AND C)
4. NOT(A OR B) = NOT(A) AND NOT (B)

Las reglas a seguir para las relaciones son bastante fáciles de entender pero las de la lógica son ya menos asequibles. La primera regla lógica dice que NOT dos veces, no cambia nada. La segunda y tercera reglas lógicas se entienden más fácilmente comparándolas con la regla aritmética:

A*(B + C) = (A*B) + (A*C)

Así, tanto la AND como la OR pueden ser factores exactamente igual que en la multiplicación.

La regla final es la más difícil. Recibe el nombre de ley de De-Morgan. Es importante porque relaciona la AND y la OR en el sentido de que cualquier expresión con una OR se puede convertir en otra con una AND y viceversa. No es ésta una lista completa de reglas para la manipulación de la lógica, pero sí sirve para salir airoso de la mayoría de los problemas lógicos.

Los restantes operadores lógicos

Los AND, OR y NOT son los únicos operadores lógicos que se necesitan, pero algunas veces es útil dar nombres a las combinaciones

de los mismos que más se utilizan. Es más, no muchas versiones de BASIC ofrecen más que los AND, OR y NOT. Luego es importante saber componer y descomponer estos operadores más complicados.

NOT(A OR B) = A NOR B
NOT(A AND B) = A NAND B
(NOT(A) AND B) OR (A AND NOT (B)) = A EOR B

Los dos primeros ejemplos son fáciles; el tercero, EOR, parece imposible. De hecho, EOR —OR Exclusiva— es uno de los operadores lógicos más útiles. Una tabla de verdad servirá de ayuda para averiguar lo que hace.

A	B	A EOR B
0	0	0
1	0	1
0	1	1
1	1	0

Observando esta columna se debe captar una similitud con la tabla de verdad correspondiente a OR. La diferencia estriba en que EOR es cierta (es 1) sólo cuando A o B, una de ellas, es cierta, y OR es cierta si una o ambas son ciertas.

La electrónica y la lógica

Al principio de este capítulo se explicaba que la lógica se utilizaba en el diseño (y depuración) de circuitos digitales. Esta sección se ocupa de la conexión entre la programación y la lógica del soporte físico (hardware).

Los circuitos digitales están compuestos por una variedad de circuitos integrados que proporcionen las funciones lógicas elementales. Por ejemplo, el circuito integrado 7400, de la gama 74, contiene cuatro puertas NAND. Hace lo mismo que la expresión NOT(A AND B) dada en la última sección, con la sola diferencia de que, en vez de ser representados los estados cierto y falso por un 1 y un 0, se utilizan dos niveles de tensión (5V y 0V), y A y B son

dos entradas al circuito. Luego, si se tiene una red de circuitos integrados lógicos sencillos, se puede escribir una expresión Booleana que los represente y utilizar un programa, basado en el dado anteriormente, para calcular una tabla de verdad y saber por lo tanto qué salida corresponde a cada entrada. Esto es, sin duda, mucho más fácil que construir el circuito.

La idea de convertir un elemento electrónico digital en una expresión Booleana es la base de una técnica muy importante que puede servir para probar cualquier diseño digital. La razón de que haya que ampliar el método estriba en que la mayoría de los equipos digitales complicados van más allá de la lógica «estática» al emplear osciladores y generadores de impulsos para «procesar» los circuitos lógicos. Hasta ahora no se ha explicado en este libro la forma de generar tales impulsos, o relojes según la denominación más normal, pero no es difícil y constituye un proyecto muy interesante que los lectores pueden abordar.

Campo de evaluación

Esta sección es un poco más avanzada que el resto del capítulo por lo que el que quiera puede saltársela y volver a ella más tarde. ¿Qué pasa si se intenta procesar algo como lo que se indica a continuación?

```
10 A = 6
20 B = 7
30 PRINT A AND B
```

Hasta aquí se han analizado solo operaciones lógicas que implican valores como 0 ó 1. Lo que pasa cuando se intenta hacer la operación lógica AND con números como 6 ó 7 depende de la versión de BASIC usada. Algunas tratan los números como números binarios (es decir, 6 = 110 y 7 = 111) y efectúan las operaciones lógicas con las parejas correspondientes de bits. Así, el bit uno del 6 es el 0 y el bit uno del 7 es el 1. Luego el resultado de la operación AND es 0 y así con el segundo y el tercer par de bits, dando la respuesta 110. El resultado de esta operación lógica con bits

puede ser tratado como otro número binario y convertido después en decimal (110 es el 6). Cualquier BASIC que opere de esta forma tendrá un número máximo con el que pueda operar. Este número máximo es el enumerado en la tabla del apartado «Variables lógicas», en la columna «Campo de evaluación».

El ZX81 y el ZX Spectrum son muy especiales en este tema del empleo de las operaciones lógicas con campos ampliados (es decir, además del 0 y el 1). Si se intenta pasar

```
10 PRINT A AND B
```

se verá que si B es cero el resultado es cero, pero si B es distinto el resultado es A. Esta notable característica es aplicable también a las cadenas.

```
10 PRINT A$ AND B
```

no imprimirá nada (imprimirá la cadena nula) si B es cero y A$ si B es cualquier otro valor. Esta característica es particularmente útil en el caso de intentar adecuar un programa al ZX81 1K o al ZX Spectrum de 16K, porque en vez de la línea

```
10 IF B<>0 THEN PRINT A$
```

se puede usar la línea

```
10 PRINT A$ AND B
```

Los otros operadores lógicos del ZX81 y del ZX Spectrum son también especiales: A OR B es A si B es cero, y 1 si B es cualquier otra cosa; NOT A es 0 si A no es cero, y 1 si lo es. Ninguno de ellos se puede utilizar con cadenas pero hay instrucciones de grandes posibilidades que merecen un estudio aparte. No obstante, corresponderían al encabezamiento «programación muy avanzada» y están fuera del alcance de este libro.

Preguntas de autoevaluación

1. Calcúlense las tablas de verdad correspondientes a las siguientes expresiones:

(a) A AND B AND C
(b) (A AND B) OR (A AND C)
(c) A AND (B OR C)

2. Escríbase una sentencia IF única que detecte el hecho de que el valor de una variable está comprendido en el campo de L a H inclusive.

3. Escríbase un programa que calcule

Q = NOT(R AND NOT(Q AND S))

para todos los valores de R, S y Q. ¿Puede reconocer el bien conocido circuito digital que esta expresión representa?

8
PRACTICAS DE ELECTRONICA CON EL BASIC

Este capítulo es un conjunto de programas-ejemplo escritos en BASIC. Ninguno de ellos es muy largo porque esto no se correspondería con el objetivo de presentación de ejemplos, pero todos ellos son completos en el sentido de que cubren la misión que tienen encomendada sin dejar puntos al aire. A diferencia, por lo tanto, de los programas del Capítulo 6, son ejemplos de programas acabados y conceptuados como tales deben ser estudiados.

Diseño de un regulador de diodo Zener

Un circuito muy común es el regulador de diodo Zener (ver figura 8.1). En la mayoría de los casos el diseño de estos reguladores se hace o adivinando el valor de la resistencia o bien realizando el cálculo para una corriente de carga concreta y esperando que todo funcione bien para otros valores. Es muy fácil escribir un programa de microordenador que calcule un adecuado valor de R y que diga la disipación de potencia en la resistencia y en el diodo. Si se conoce la tensión de entrada mínima, la corriente de carga máxima y la tensión en el diodo Zener, un valor razonable de R es:

$$R = \frac{\text{V entrada máx. — V Zener}}{\text{corriente carga máx.}}$$

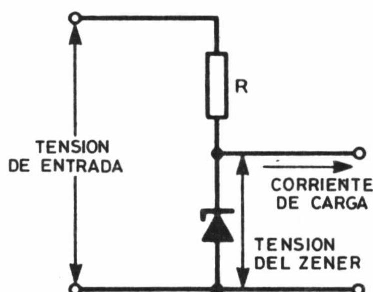


Figura 8.1. Regulador de diodo Zener.

Esta ecuación se ha establecido considerando la peor condición posible, por ejemplo, cuando la corriente de carga es alta y la tensión de alimentación es baja, y calculando la resistencia necesaria en tal situación. (La tensión entre bornes de la resistencia es siempre la tensión de entrada menos la tensión entre bornes del diodo Zener.) Hay una afirmación gratuita en este cálculo: que el diodo Zener no disipa corriente. En la práctica, el diodo Zener sí dispa corriente y esa corriente que disipa ha de ser añadida a la corriente de carga máxima. Por esta razón, la ecuación da un valor de R que es ligeramente más alto. Una vez calculado el valor de R, el cálculo de la disipación de potencia en el caso peor es fácil:

$$\text{Potencia en la resistencia máx.} = \frac{(\text{V entrada máx.} - \text{V Zener})^2}{\text{R}}$$

Potencia en Zener máx. =

$$\text{V Zener x} \left[\frac{(\text{V entrada máx.} - \text{V Zener})}{\text{R}} - \text{Corriente carga mín.} \right]$$

Con estas sencillas ecuaciones y los conocimientos de BASIC adquiridos, no es difícil ejecutar el programa de diseño del circuito de diodo Zener siguiente:

```
 10 PRINT "TENSION CC MAXIMA ";
 20 INPUT V
 30 PRINT "TENSION CC MINIMA ";
 40 INPUT W
 50 PRINT "TENSION DIODO ZENER
";
 60 INPUT Z
 70 PRINT "CORRIENTE DE CARGA M
AXIMA ";
 80 INPUT J
 90 PRINT "CORRIENTE DE CARGA M
INIMA ";
100 INPUT I
110 LET R=(V-Z)/J
120 LET P=(V-Z)*(V-Z)/R
130 LET Q=((V-Z)/R-I)*Z
140 PRINT "VALOR APROX. DE LA R
ESISTENCIA","=";R
150 PRINT "POTENCIA MAX. EN LA
RESISTENCIA","=";P
160 PRINT "POTENCIA MAX. EN EL
ZENER","=";Q
```

Aunque este programa es muy sencillo es a la vez muy útil y en un momento podría hacerse de él un programa de análisis del diodo Zener completo, incluyendo características como dar al usuario la posibilidad de hacer pruebas con una gama de valores de R y de calcular el factor de regulación. Por ejemplo, el factor de regulación (la proporción entre la variación de los voltios de salida con una determinada variación de los voltios de entrada) viene dada por

$$1/(1 + R/RL + R/RZ)$$

donde RL es la resistencia de carga (constante) y RZ es la resistencia de pendiente del diodo Zener. Añadiendo esta parte al programa anterior queda así:

```
170 PRINT "RESISTENCIA  DE  PEN
DIENTE  DEL  ZENER ";
180 INPUT S
190 LET A=Z/J
200 LET B=Z/I
210 PRINT "CARGA","REGULACION"
220 FOR K=B TO A STEP (A-B)/20
230 PRINT K,1/(1+R/K+R/S)
240 NEXT K
```

Hay que destacar que la estimación de la gama de resistencias de carga se realiza utilizando las corrientes máxima y mínima

y la tensión en el diodo Zener. También hay que señalar la forma de componer la tabla de factores de estabilización, a base de 20 intervalos igualmente espaciados, con independencia de la gama de resistencia de carga.

El puente de Wien

En los osciladores de ondas senoidales de baja frecuencia se suele utilizar la disposición de elementos del puente de Wien (ver figura 8.2). La ecuación que da la impedancia del puente propiamente dicho es muy fácil de obtener a partir de la figura 8.3 usando impedancias complejas. El puente propiamente dicho es simplemente un divisor de tensión donde

$$Z1 = R1 - j/\omega C1 \qquad \text{(impedancia en serie)}$$

y

$$\frac{1}{Z2} = \frac{1}{R2} + j\omega C2 \qquad \text{(impedancias en paralelo)}$$

Es decir,

$$Z2 = \frac{1}{1 + j\omega C2R2}$$

Utilizando la fórmula normal de las tensiones de un divisor de tensión se obtiene

$$\frac{VI}{VO} = \frac{Z1 + Z2}{Z2}$$

$$= 1 + \frac{(R1 + (1/j\omega C1))\,(1 + j\omega C2R2)}{R2}$$

$$= 1 + \frac{R1}{R2} + \frac{C2}{C1} + j(\omega R1C2 - \frac{1}{\omega R2C1})$$

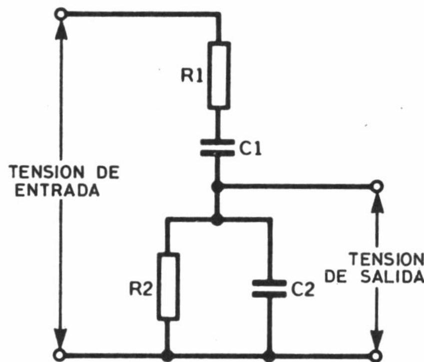


Figura 8.2. Puente de Wien.

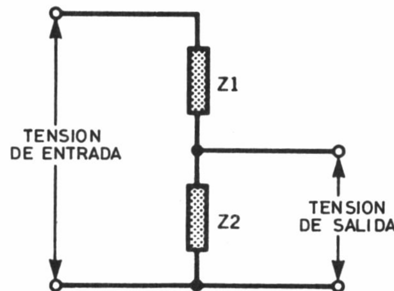


Figura 8.3. Circuito divisor de tensión equivalente.

Esta última ecuación da la respuesta del puente de Wien en forma cartesiana. A partir de aquí, se podrían utilizar los métodos introducidos en el Capítulo 6 para trazar la amplitud y el ángulo de fase de la respuesta. Sin embargo, tratándose de un oscilador, el desplazamiento de fase, introducido por el puente de Wien, debe ser precisamente cero. (El efecto de la realimentación positiva viene del propio amplificador). (Véase figura 8.4). Esto corresponde a la desaparición de la parte imaginaria de la respuesta, dando un ángulo de fase cero. Esto implica que a la frecuencia de oscilación:

$$\omega R1C2 + \frac{1}{\omega R2C1}$$

En otras palabras:

$$\omega = \frac{1}{\sqrt{(R1R2C1C2)}}$$

A efectos de simplificación de los cálculos se suele partir generalmente del supuesto de que R1 = R2 y C1 = C2, resultando que la frecuencia es

$$\omega = 1/RC$$

donde ω es la frecuencia angular: $\omega = 2\pi f$

Con esta sencilla ecuación se pueden calcular los valores de los componentes para una determinada frecuencia. Es normal seleccionar un valor de R y luego calcular el valor de C necesario para la frecuencia deseada. Si este valor de C no es conveniente, o no se puede obtener, se cambia entonces el valor de R se altera y se repite el cálculo. El siguiente programa facilita esta repetición:

```
10 PRINT "PUENTE DE WIEN"
20 PRINT
30 PRINT "TODAS LAS RESISTENCI
```

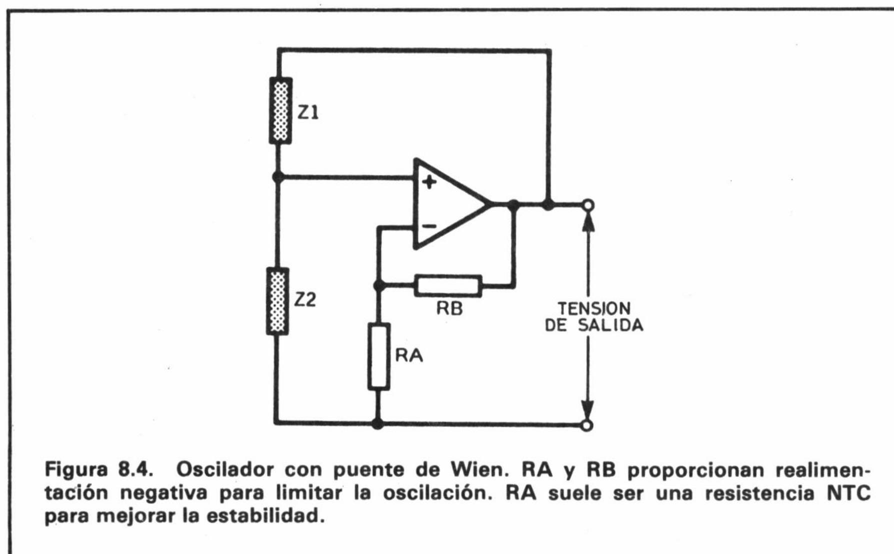


Figura 8.4. Oscilador con puente de Wien. RA y RB proporcionan realimentación negativa para limitar la oscilación. RA suele ser una resistencia NTC para mejorar la estabilidad.

```
AS EN K Y LOS CONDENSADORES EN N
F"
  40 PRINT
  50 PRINT
  60 PRINT "FRECUENCIA DE RESONA
NCIA EN HZ =";
  70 INPUT F
  80 PRINT "VALOR DE LA RESISTEN
CIA =";
  90 INPUT R
 100 LET R=R*1000
 110 LET C=1/(2*3.14159*R*F)
 120 PRINT "CAPACIDAD NECESARIA
= ";C*1E9;" NF"
 130 GO TO 10
```

El único problema que presenta el diseño de este programa es que arrastra la presunción tradicional de hacer R1 = R2 y C1 = C2, un caso raro en la práctica por la tolerancia de los componentes. Un segundo problema estriba en la dificultad para obtener una frecuencia exacta si se utilizan valores preferentes de los componentes. Cuando se emplea un microordenador parece que no hay razones para no explorar la posibilidad de utilizar diferentes valores para cada componente. El siguiente programa calcula la frecuencia de resonancia de un puente de Wien.

```
  10 PRINT "PUENTE DE WIEN"
  20 PRINT
  30 PRINT "TODAS LAS RESISTENCI
AS EN K Y LOS CONDENSADORES EN N
F"
  40 PRINT
  50 PRINT
  60 PRINT "R1= ";
  70 INPUT R
  80 LET R=R*1000
  90 PRINT "R2= ";
 100 INPUT S
 110 LET S=S*1000
 120 PRINT "C1= ";
 130 INPUT C
 140 LET C=C*1E-9
 150 PRINT "C2= ";
 160 INPUT D
 170 LET D=D*1E-9
 180 LET W=SQR (1/(R*S*C*D))
 190 LET F=W/(2*3.14159)
 200 PRINT "FRECUENCIA = ";F;" H
Z"
 210 GO TO 10
```

Si se utiliza este programa en conjunción con el primero, es posible estudiar los efectos de la introducción de ligeros cambios

en los valores de los componentes y los efectos de adoptar el siguiente valor preferente, superior e inferior. Así se puede optimizar el funcionamiento del puente por el método de pruebas y errores.

La idea de utilizar un microordenador como herramienta para la búsqueda de una solución es bastante común en las aplicaciones de diseño de frecuencias. Sería posible escribir un programa mucho más completo, que averiguara la solución óptima directamente pero, a menos que haya que diseñar frecuentemente puentes de Wien, esto daría más trabajo de lo que la aplicación merece.

Aunque los dos programas presentados anteriormente son buenas herramientas para calcular puentes de Wien, es posible que dejen la sensación de que puede haber efectos secundarios debidos a la no utilización de un puente donde R1 = R2 y C1 = C2. Una forma de examinar tales efectos secundarios consiste en analizar la forma en que varía la respuesta de frecuencia del puente al ajustar los valores de los componentes. Cabría la posibilidad de usar el programa dado en el Capítulo 6 para trazar la gráfica de la impedancia pero, dado que se trata de una necesidad muy común, puede merecer la pena ver si hay alguna posibilidad de hacer algo más general.

Trazado de la gráfica de la amplitud y de la fase

Como ya conocemos un método de trazar gráficos con el microordenador, pudiera parecer que no hay nada más que añadir. Sin embargo, en el programa siguiente de trazado de gráficas se pone el acento en la escritura de un programa de tipo general que sirva para presentar visualmente cualquier función compleja en términos de la amplitud y de la fase. Así, cualquier cambio que haya que hacer para trazar la gráfica de una función diferente quedará reducido a subrutinas individuales. Además, para que ofrezca un valor práctico, la amplitud y la fase deben estar presentadas en la misma gráfica y se deben incluir las escalas. Se crea o no, es este último requisito el que es más difícil de cumplir. Se necesita un ser humano para determinar una escala de una forma adecuada. Los microordenadores suelen acabar dando intervalos fraccionarios y un número de divisiones absurdo.

Por ejemplo, una persona dividiría una escala de 0 a 100 en diez pasos iguales o, si fuera demasiado fino, en cinco pasos iguales, pero la mayoría de las rutinas de determinación automática de escalas probablemente dividirían ese campo en seis o cuatro pasos iguales, por no decir cualquier otro número.

```
  10 PRINT "TRAZADO DE LA GRAFIC
A DE LA FRECUENCIA"
  20 PRINT
  30 GO SUB 1000
  40 GO SUB 2000
  50 GO SUB 3000
  60 PRINT "NUEVOS VALORES COMPO
NENTES O NUEVO FACTOR ESCALA (C/
S);"
  70 INPUT A$
  80 IF A$="C" THEN GO TO 30
  90 IF A$="S" THEN GO TO 40
 100 GO TO 60
1000 PRINT "R1= ";
1010 INPUT R
1020 LET R=R*1000
1030 PRINT "R2= ";
1040 INPUT S
1050 LET S=S*1000
1060 PRINT "C1= ";
1070 INPUT C
1080 LET C=C*1E-9
1090 PRINT "C2= ";
1100 INPUT D
1110 LET D=D*1E-9
1120 LET L=40
1130 RETURN
2000 PRINT
2010 PRINT "TRAZAR LA GAMA DE FR
ECUENCIAS"
2020 PRINT "COMENZANDO POR ";
2030 INPUT B
2040 PRINT "TERMINANDO POR ";
2050 INPUT E
2060 IF B>E THEN GO TO 2000
2070 PRINT
2080 PRINT "FACTOR DE ESCALA (IM
PEDANCIA MAYOR)";
2090 INPUT M
2100 RETURN
3000 FOR I=0 TO M STEP M/10
3010 PRINT TAB (I*L/M+5);I;
3020 NEXT I
3030 PRINT
3040 FOR I=0 TO M STEP M/10
3050 PRINT TAB (I*L/M+6);"+";
3060 NEXT I
3070 PRINT
3080 FOR F=B TO E STEP (E-B)/20
3090 PRINT F;
3100 LET W=2*3.14159*F
```

```
3110 GO SUB 4000
3120 IF I>L THEN LET I=L
3130 IF J>L THEN LET J=L
3140 IF I<J THEN PRINT TAB (I);"
X";TAB (J);"*"
3150 IF J<I THEN PRINT TAB (J);"
*";TAB (I);"X"
3160 IF I=J THEN PRINT TAB (I);"
"
3170 NEXT F
3180 PRINT TAB (6);"+";
3190 PRINT TAB (L/2+6);"+";
3200 PRINT TAB (L+6);"+"
3210 PRINT TAB (5);"-PI";
3220 PRINT TAB (L/2+5);"0";
3230 PRINT TAB (L+5);"PI"
3240 RETURN
4000 GO SUB 9000
4010 LET A=SQR (ZR*ZR+ZI*ZI)
4020 LET P=ATN (ZI/ZR)
4030 LET I=INT (A*L/M+6.5)
4040 LET J=INT ((3.14159+P)/2/3.
14159*L+6.5)
4050 RETURN
9000 LET ZR=1+R/S+C/D
9010 LET ZI=W*R*D-1/(W*S*C)
9020 RETURN
```

He aquí un programa que traza la respuesta de frecuencia. El programa principal, líneas 10 a 100, tiene la forma de un bucle que llama a las subrutinas apropiadas cuando son necesarias. La subrutina 1000 sirve para leer los valores de los componentes. Esta subrutina tendría que ser modificada para hacer el trazado de una ecuación diferente. La subrutina 2000 lee el campo de frecuencias para el que va a ser trazada la respuesta y el factor de escala, es decir, el valor de impedancia más alto a incluir en la gráfica.

La subrutina 3000 tiene a su cargo la función de trazado de la gráfica. Las líneas 3000 a 3070 se encargan de generar la primera escala horizontal correspondiente a la amplitud. Esta escala está simplemente dividida en diez espacios iguales comprendidos entre 0 y M, el valor más grande que se ha de incluir en la gráfica. Las líneas 3080 a 3170 generan la gráfica y hacen una llamada a la subrutina 4000 para obtener los valores de I y J que dan las posiciones de las «X» y de los «*» respectivamente. La amplitud se traza mediante símbolos «X» y la fase mediante signos «*». Hay que destacar que la subrutina 4000 en vez de dar los valores reales de la amplitud y de la fase los pone a escala y da la posición de línea donde deben salir los símbolos. Para trazar ambos símbolos

en la misma línea, está claro, que es preciso imprimir primero uno a la izquierda del todo ya que la función TAB no puede mover la posición de impresión hacia atrás, efecto que se consigue mediante las líneas 3140 y 3150. La línea 3160 se ocupa del caso de coincidencia de la "X" y el "*" en el mismo lugar, y las líneas 3120 y 3130 garantizan que no se sobrepasará la anchura de línea. La parte final de la subrutina 3000 se limita a imprimir un segundo eje horizontal, esta vez señalizado, mostrando la fase.

La subrutina 4000 calcula las I y las J a partir de las partes real e imaginaria de los valores almacenados en ZR y ZI respectivamente. Estos valores son calculados por la subrutina 9000 a partir de la fórmula de la respuesta de frecuencia. Esta subrutina 9000 y la subrutina 1000 son las dos únicas subrutinas que hay que cambiar para trazar una respuesta de frecuencia distinta.

Finalmente, el programa principal acaba con una posibilidad de elección entre volver a trazar la gráfica de la respuesta con valores distintos de los componentes o simplemente con un factor de escala distinto. Esto permite al usuario fijar un factor de escala por el método de la prueba y el error. Hay que destacar también que el usuario puede conseguir tener un conjunto de ejes limpio, mediante una cuidada selección de la gama de frecuencias a trazar.

El punto más importante contenido en este ejemplo es la forma en que las subrutinas permiten hacer un programa más fácil de escribir y de entender.

Ecuaciones múltiples y redes

El cálculo de las corrientes y de las tensiones de una red compuesta por resistencias y baterías es sencillamente una cuestión de aplicar las leyes de Kirchhoff:

> *La corriente total que sale de un nudo es igual a la corriente total que entra en el nudo;*

y la ley de la tensión:

> *La fem de un circuito es igual a la suma de las caídas de tensión que hay en el circuito.*

En el caso del circuito mostrado en la figura 8.5 las leyes de Kirchhoff dan

$$I1 + I2 + I3 = 0$$
$$V1 = R1*I1 + R3(I1 + I2)$$

y

$$V2 = R2*I2 + R3(I1 + I2)$$

Normalmente conocemos los valores de V1, V2, R1 y R2, y si se quieren conocer las corrientes que circulan por las tres resistencias, es decir, I1, I2 e I3. Si R1 = 500 ohmios, R2 = 200 ohmios y R3 = 300 ohmios, V1 = 6V y V2 = 3V, se pueden escribir de nuevo las ecuaciones así:

$$\begin{aligned} I1 + I2 + I3 &= 0 \\ 800I1 + 300I2 + 0I3 &= 6 \\ 300I1 + 500I2 + 0I3 &= 3 \end{aligned}$$

Para obtener las intensidades hay que resolver este conjunto de ecuaciones múltiples. El método más generalizado consiste en restar

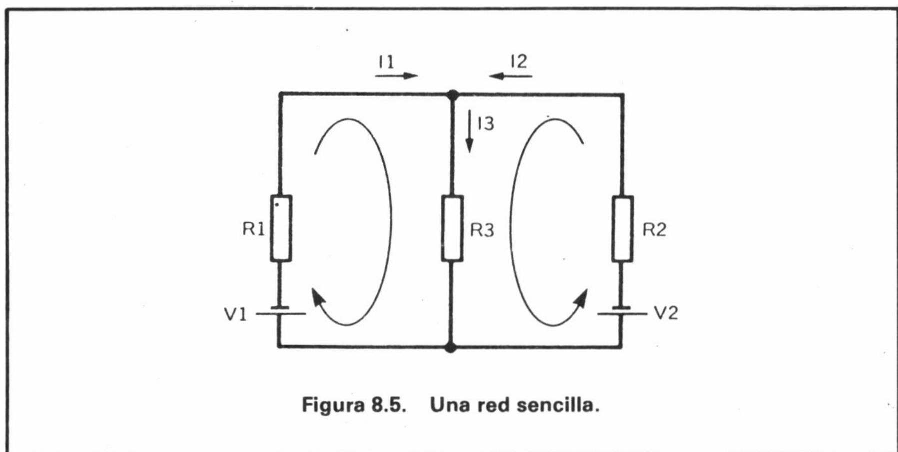


Figura 8.5. Una red sencilla.

múltiplos de una fila (de una ecuación) con respecto a otra hasta reducir las ecuaciones a una forma que tenga coeficientes de valor 1 en la diagonal y ceros por debajo. Por ejemplo, restando 800 veces la primera ecuación de la segunda da:

$$
\begin{aligned}
I1 + I2 + I3 &= 0 \\
-500I2 - 800I3 &= 6 \\
300I1 + 500I2 + 0I3 &= 3
\end{aligned}
$$

Restando 300 veces la primera ecuación de la tercera da:

$$
\begin{aligned}
I1 + I2 + I3 &= 0 \\
-500I2 - 800I3 &= 6 \\
-200I2 - 300I3 &= 3
\end{aligned}
$$

Se han eliminado así los términos en I1 aparte del de la primera ecuación. Lo que se hace con este proceso es eliminar la primera variable de la segunda fila y de la tercera, utilizando la primera ecuación. En términos generales, apoyándose en la ecuación enésima se puede eliminar la variable enésima de todas las ecuaciones siguientes. Por ejemplo, para quitar la segunda variable de la tercera ecuación lo único que hay que hacer es apoyarse en la segunda ecuación; el único problema es que el coeficiente de la segunda variable de esta segunda ecuación no es un 1 y esto dificulta el cálculo del factor por el que hay que multiplicarla. Ese problema se resuelve multiplicando la segunda ecuación por 1/—500, resultando

$$
\begin{aligned}
I1 + I2 + I3 &= 0 \\
I2 + 1.6I3 &= -0.012 \\
200I2 - 300I3 &= 3
\end{aligned}
$$

Sustrayendo ahora 200 veces la segunda ecuación de la tercera tendremos:

$$
\begin{aligned}
I1 + I2 + I3 &= 0 \\
I2 + 1.6I3 &= -0.012 \\
-620I3 &= 5.4
\end{aligned}
$$

Ahora se puede obtener fácilmente la solución a las ecuaciones.

Por la tercera ecuación sale que I3 = —5.4/620, aproximadamente —9mA. Utilizando este resultado en la segunda ecuación se obtiene:

I2 = —0.012 — 1.6I3

aproximadamente 2 mA. Finalmente, pasando ambos resultados a la primera ecuación se obtiene una solución de aproximadamente 7 mA para I3.

Así, la solución de cualquier conjunto de ecuaciones se puede considerar como la reducción de las variables de cada ecuación paso a paso y la aplicación de la sustitución hacia atrás. Escribir un programa que realice estas operaciones es bastante sencillo pero antes merece la pena resumir las operaciones que implica la reducción de la fila I:

1. Dividir cada coeficiente de la fila I por el coeficiente I de esta fila. (Esto reduce el coeficiente I a 1).
2. Restar de la fila I + 1 la nueva fila I multiplicada por el coeficiente I de la fila I + 1.
3. Repetir el paso 2 para la fila I + 2, la fila I + 3, etc., hasta llegar a la última fila, donde la reducción es total.

Lo único que hay que hacer para la solución de la substitución final es repetir esta operación de reducción para I = 1 hasta N (I = 1 TO N), donde N es el número de ecuaciones.

```
  10 PRINT "RESOLUCION DE ECUACI
ONES MULTIPLES"
  20 PRINT "NUMERO DE ECUACIONES
 ";
  30 INPUT N
  40 DIM C(N,N+1)
  50 GO SUB 1000
  60 GO SUB 2000
  70 GO SUB 4000
  80 GO SUB 5000
  90 GO SUB 6000
 100 STOP
1000 FOR I=1 TO N
1010 PRINT "COEFICIENTES PARA EC
UACION ";I
1020 FOR J=1 TO N
1030 INPUT C(I,J)
1040 NEXT J
1050 PRINT "CONSTANTE =";
1060 INPUT C(I,N+1)
1070 NEXT I
```

```
1080 PRINT
1090 RETURN
2000 FOR I=1 TO N-1
2010 GO SUB 3000
2020 NEXT I
2030 LET C(N,N+1)=C(N,N+1)/C(N,N
)
2040 LET C(N,N)=1
2050 RETURN
3000 LET C=C(I,I)
3010 FOR J=1 TO N+1
3020 LET C(I,J)=C(I,J)/C
3030 NEXT J
3040 FOR J=I+1 TO N
3050 LET C=C(J,I)
3060 FOR K=1 TO N+1
3070 LET C(J,K)=C(J,K)-C(I,K)*C
3080 NEXT K
3090 NEXT J
3100 RETURN
4000 FOR I=1 TO N
4010 FOR J=1 TO N+1
4020 PRINT TAB (J*10);C(I,J);
4030 NEXT J
4040 PRINT
4050 NEXT I
4060 RETURN
5000 DIM A(N)
5010 LET A(N)=C(N,N+1)
5020 FOR I=N-1 TO 1 STEP -1
5025 LET A(I)=C(I,N+1)
5030 FOR J=I+1 TO N
5040 LET A(I)=A(I)-A(J)*C(I,J)
5050 NEXT J
5060 NEXT I
5070 RETURN
6000 FOR I=1 TO N
6010 PRINT "VARIABLE ";I;" = ";A
(I)
6020 NEXT I
6030 RETURN
```

El programa dado aquí resolverá cualquier número de ecuaciones múltiples por el método de la reducción. El programa principal está constituido por la lista acostumbrada de llamadas a subrutinas, aparte de las líneas 10 a 40 que definen una matriz C (N por N + 1). La subrutina 1000 lee los coeficientes de cada ecuación uno por uno y los almacena en C. No se da al usuario ninguna oportunidad de corregir errores, así que ¡cuidado al teclear!. La subrutina 2000 lleva a cabo la operación de reducción de cada una de las filas una por una. El bucle FOR de las líneas 2000 a 2020 lleva a cabo una operación de reducción en la fila I, para las filas 1 a N—1 (1 TO N —1), mediante una llamada a la subrutina 3000. Las

líneas 2030 a 2040 reducen el trabajo de la sustitución final cambiando el coeficiente de la ecuación final a 1 después de finalizadas todas las operaciones de reducción. La subrutina 3000 realiza la reducción de la línea I, y tal como se ha dicho, es utilizada repetidas veces por la subrutina 2000. Las líneas 3000 a 3030 modifican la fila I de manera que C(I,I) sea 1, es decir, realiza el paso 1 de la reducción.

El resto de esta subrutina tiene la forma de un par de bucles FOR anidados. El bucle interior (líneas 3060 a 3080) resta el múltiplo correcto de la fila I con respecto a la fila J, donde J varía desde I + 1 a N, que determina el bucle exterior (líneas 3040 a 3090). Cuando esta sustitución llega a su fin, la variable I ha sido eliminada de todas las ecuaciones I + 1 a N.

La subrutina 4000 imprime la versión final de las ecuaciones, listas para la sustitución final, y la subrutina 5000 lleva a cabo esta sustitución final. Por último, la subrutina 6000 imprime las respuestas que están almacenadas en la matriz A.

Esto es todo para la resolución de ecuaciones simultáneas por microordenador. Cabe señalar que no se ha dado ningún detalle relativo a la subrutina de sustitución final. La razón está en que el lema de esta sección es: todo lo que se sepa calcular a mano debe saber convertirse en un programa de microordenador. Teniendo esto en cuenta, debe resultar fácil desentrañar la subrutina 5000. Si se encontraran dificultades, estudie el funcionamiento del factor multiplicador a la hora de hacer la sustitución a mano.

Si se teclea los coeficientes de las ecuaciones múltiples dadas anteriormente se verá que se imprimen las soluciones exactas sin necesidad de hacer nada más. Pero si se hace la prueba con un conjunto de ecuaciones como

$$I1 + 0I2 + 3I3 = 0$$
$$4I1 + 0I2 + 4I3 = 12$$
$$I1 - 3I3 - I3 = 6$$

se verá que el programa no da una solución sino que se para en la línea 3020. La causa de ello es que se está intentando hacer la reducción en la segunda ecuación, donde el segundo coeficiente es cero y la división por cero es indefinida. La solución está en reor-

ganizar el orden de las ecuaciones de manera que la segunda ecuación se convierta en la tercera. Este tipo de problemas puede surgir en cualquier momento a la hora de resolver ecuaciones múltiples por el método de la reducción.

Otro problema conexo es que la precisión del método depende del tamaño del coeficiente por el que se divide: cuanto mayor es más exacto es el resultado, y viceversa. La solución obvia a estos problemas consiste en reorganizar la matriz de manera que el elemento mayor esté en C(I,I), justo antes de la reducción de la fila I. La incorporación de esta operación al programa resulta fácil por el hecho de que está escrito a base de subrutinas. El único cambio con respecto al programa existente es añadir la línea 2005,

```
2005 GOSUB 7000
```

a la subrutina 2000. La nueva subrutina, que comienza en la línea 7000, comprueba los elementos de la columna I debajo de C(I,I). Si encuentra uno mayor intercambia las dos filas. Con esta incorporación el programa resolverá ecuaciones simultáneas que tengan solución.

```
7000  LET M=C(I,I)
7010 LET K=I
7020 FOR J=I TO N
7030 IF ABS (M)>ABS (C(N,I)) THE
N GO TO 7040
7035 LET K=J
7036 LET M=C(J,I)
7040 NEXT J
7050 IF K=I THEN RETURN
7060 FOR J=1 TO N+1
7070 LET T=C(I,J)
7080 LET C(I,J)=C(K,J)
7090 LET C(K,J)=T
7100 NEXT J
7110 RETURN
```

La descomposición de programas largos en subrutinas

Los dos últimos programas de este capítulo, el de trazado de gráficas y el de resolución de ecuaciones múltiples, se consideran bastante largos y difíciles. No obstante, si se descompone un programa largo en subrutinas se le podrá atacar como si fuera una serie

de programas pequeños. Mirado de esta forma, los programas largos no ofrecen dificultades nuevas y ahora el lector ya debe ser capaz de escribir por sí solo programas útiles. Sin embargo, hay que tener siempre presente que el microordenador no puede hacer un cálculo, o lo que sea, que el programador no sepa hacer a mano, al menos en principio. Como en el ejemplo de las ecuaciones múltiples, lo más útil es resolver un ejemplo a mano (o con calculadora) para estar seguro de que se entiende el método seguido. Esto tiene la ventaja de que se dispone ya de un caso concreto aplicable a ese programa, del que se conoce la respuesta.

Antes de empezar un programa, lo mejor es descomponer siempre en pequeños fragmentos lo que se está haciendo. Se puede decir que la divisa del programador es : «divide y vencerás».

9

PROGRAMACION AVANZADA EN BASIC

No hay que alarmarse por el título de este capítulo porque, en este caso, «avanzado» no significa difícil. Para ser más exactos, este capítulo se ocupa de las características no-estándar del BASIC. Hasta ahora hemos utilizado únicamente esas partes del BASIC que normalmente se pueden encontrar en cualquier microordenador. La ventaja de no salirse de los límites marcados por este subconjunto de BASIC radica en que los programas que se escriban se pueden transferir fácilmente a otro microordenador. Hay razones, sin embargo, para saber algo sobre las características especiales que cabe encontrar en algunos microordenadores. Algunas veces, la única forma de lograr que un programa se ejecute en un microordenador concreto es utilizar algunas instrucciones especiales. También es conveniente conocer las limitaciones del microordenador que poseemos averiguando qué es lo que le falta.

Indicadores de entrada

Si se vuelve a alguno de los programas-ejemplo anteriores se advertirá que las instrucciones INPUT suelen ir precedidas por instrucciones PRINT. Normalmente primero se imprime un mensaje y después se espera la respuesta. Para ahorrar espacio, algunas de las

versiones de BASIC permiten incluir una cadena literal en una INPUT. Por ejemplo:

```
10 INPUT "QUE EDAD TIENES", A
```

imprime QUE EDAD TIENES? y después espera una respuesta a introducir por el teclado. La cadena literal recibe el nombre de indicador de entrada. El problema de usar esta cómoda forma de INPUT radica en que, incluso las versiones de BASIC que la tipifican, no concuerdan en una misma forma exacta. Algunos usan un punto y coma en vez de una coma para separar el indicador de la variable, algunos imprimen un signo de interrogación y otros no.

El formateado de la impresión

Ya se ha visto que se puede utilizar la función TAB en combinación con un punto y coma y una coma para controlar el lugar de la pantalla en el que saldrán impresas las cosas pero, hasta ahora, no se ha puesto al descubierto ninguna forma de controlar cuántas cosas han de imprimirse en la pantalla. Por ejemplo, si se pone «PRINT A» el resultado puede salir así:.3; o así: .3333333333 o así: 0.333. No se dispone de ninguna forma de control del número de cifras decimales que se van a imprimir o del número de ceros delanteros que van a utilizarse.

El problema está resuelto en el BASIC Microsoft por la instrucción PRINT USING, cuyo formato es:

```
PRINT USING "cadena formato";lista a imprimir
```

La cadena formato puede ser o una cadena literal o una variable de la cadena y define la forma de impresión de la lista a imprimir. La gama de formatos permitidos depende, en gran medida, de cada microordenador y es muy variada. Lo más que se puede hacer aquí es dar un ejemplo sencillo.

Se puede «dibujar» un esquema del número utilizando el signo # para representar cada dígito. Por ejemplo, ###.### en un dibujo de un número de seis dígitos con tres dígitos tras el punto decimal (la coma). Así,

```
10 PRINT USING «###.###»;.78,200.45,.3333333
```

imprimirá

```
0.780        200.450        0.33
```

Si el número dado no tiene suficientes dígitos para llenar el gráfico se añaden espacios en blanco a la derecha. Si se especifica que delante del punto decimal (la coma) vaya un dígito, entonces sale impreso siempre ese dígito, incluso si se trata de un cero.

El comando PRINT USING es una instrucción muy potente y da al usuario la posibilidad de generar pantallas llenas de números pero «en limpio». En la programación técnica, sin embargo, la necesidad frecuente y generalizada es la de escribir un número con un determinado número de cifras decimales, y esto se puede hacer sin la instruccion PRINT USING. Se puede hacer así:

```
10 DEF FNA(X,N) = INT(X*N)/N
20 PRINT FNA(3.3333,1000)
```

La función FNA(X,N) imprimirá X con un número de cifras decimales igual al número de ceros que hay en N. (Para que funcione, N debe ser 10,100,1000,100000 etc.). Si en el BASIC que se utiliza no hay ninguna instrucción de definición de funciones basta usar INT(X*1000)/1000 cada vez que sea necesario.

ELSE

La sentencia de BASIC IF tiene un truco que está a la espera de ser aprovechado. Algunas versiones de BASIC permiten una forma ampliada de IF:

```
IF condición THEN "sentencia BASIC" ELSE "sentencia
BASIC"
```

Esta forma de la sentencia IF es mucho más simétrica. Si la condición se cumple (si es cierta), se ejecuta la sentencia que viene

a continuación de THEN, y si la condición es falsa (si no se cumple), se ejecuta la sentencia BASIC que viene a continuación de ELSE, es decir, solo se ejecuta una de las dos sentencias (la que viene a continuación de THEN o la que viene a continuación de ELSE). Después, el control pasa a la siguiente sentencia (generalmente a la siguiente línea del programa). Por ejemplo:

```
10 INPUT A
20 IF A = 0 THEN PRINT "A ES CERO" ELSE PRINT "A
   NO ES CERO"
30 GOTO 10
```

es lo mismo que

```
10 INPUT A
20 IF A = 0 THEN PRINT "A ES CERO"
30 IF A < > 0 THEN PRINT "A NO ES CERO"
40 GOTO 10
```

Como se puede ver, la forma IF-THEN-ELSE es más fácil y más limpia pero no esencial.

Mientras se está en el tema de las declaraciones IF y de las comprobaciones vale la pena considerar los problemas de precisión del BASIC.

El logro de la precisión en un programa

Pruébese el siguiente programa aparentemente razonable:

```
10 I = 1
20 I = I — 1/3
30 IF I = 0 THEN STOP
40 GOTO 20
```

El significado del programa es bien sencillo: se trata de igualar I a 1 y restar 1/3 hasta que no quede nada, es decir, hasta llegar al cero. Si se recorre el programa con papel y lápiz se ob-

tendrá un valor de 2/3 para I la primera vez que se llega a la línea 30, 1/3 la segunda y 0 la tercera. Luego, el programa debe pararse tras la tercera pasada. El que haya ejecutado el programa en su microordenador es muy posible que se haya quedado estupefacto al ver que el programa da vueltas y vueltas sin llegar a ninguna parte. ¿Qué ha ido mal?

Insertando 25 PRINT I se podrá ver lo que está pasando. El problema está en que la cantidad 1/3 no se almacena con total exactitud en el microordenador. Probablemente se almacena como .33333333 que es menor que 1/3. Por eso, después de tres pasadas por el bucle, I se halla muy próximo a cero, pero no es exactamente cero. Los problemas de este tipo se pueden solventar siguiendo esta regla: no hay que comprobar nunca una *igualdad* entre números que sean el resultado de cálculos. No obstante, si sólo se utilizan números enteros (ni fracciones, ni decimales, etc.) el problema no se produce.

La solución al problema de la falta de precisión en números fraccionarios consiste en cambiar las comprobaciones de igualdades por pruebas de «aproximación». Si dos números difieren en una cierta cantidad que es aproximadamente de la misma cuantía que la precisión en los cálculos del microordenador, entonces es como si fueran iguales. Por ejemplo, pruébese:

```
10 I = 1
20 I = I — 1/3
30 IF I <.000001 THEN STOP
40 GOTO 20
```

Este programa funcionará en todos los microordenadores de una precisión superior a 0.000001. Si no, basta cambiar el número de ceros que hay detrás del punto decimal hasta que el programa funcione. Generalizando, se puede cambiar

```
10 IF A = B THEN ...
```

por

```
10 IF ABS(A — B) <.000001 THEN ...
```

Líneas de múltiples instrucciones

Muchas versiones de BASIC permiten escribir más de una sentencia en una línea, separadas entre sí por dos puntos. Así

```
10 A = 1:B = 2:IF A = 0 THEN GOTO 50
```

surte los mismos efectos que

```
10 A = 1
20 B = 2
30 IF A = 0 THEN GOTO 50
```

Esta característica es de suma utilidad siempre y cuando se tengan en cuenta dos puntos. En primer lugar, las líneas de BASIC largas pueden resultar difíciles de leer y, como los mensajes de error dan generalmente un número de línea, puede resultar difícil saber qué parte de la línea larga es la causa de problemas. Y en segundo lugar, la mayoría de las versiones de BASIC imponen un límite a la longitud de las líneas (generalmente de 80 caracteres) y esto, a su vez, limita el número de sentencias que es posible poner en una línea.

Antes de pasar a trabajar con líneas de múltiples sentencias merece la pena averiguar con más precisión a dónde envía exactamente GOSUB. Por ejemplo, ¿qué imprime en su microordenador el siguiente programa?

```
10 A = 0:GOSUB 1000:A = 1:GOSUB 1000
20 A = 2:GOSUB 1000
30 STOP
1000 PRINT A
1010 RETURN
```

Si la primera GOSUB hace un traslado a la línea siguiente (a la línea 20) saldrá en la pantalla un 0,2 pero si la primera GOSUB vuelve al punto de completar el resto de la línea de múltiples sentencias, entonces aparecerá en la pantalla la serie 0,1,2.

Una aplicación de las líneas de sentencias múltiples está en las sentencias IF-THEN-ELSE. Si se tienen dos listas, una que se

quiere ejecutar si la condición es cierta y otra si la condición es falsa, cabría la posibilidad de utilizar la forma

IF condición THEN lista 1 ELSE lista 2

donde lista 1 y lista 2 son listas de sentencias de BASIC separadas por dos puntos. No obstante, si las listas son demasiado largas, no sólo hacen difícil la lectura y depuración de la línea sino que además no hay muchas posibilidades de que el BASIC permita ponerlas todas en la misma línea. La mejor solución a este problema consiste en agrupar las listas como dos subrutinas BASIC y utilizar

IF condición THEN GOSUB lista 1 ELSE GOSUB lista 2

donde, en este caso concreto, lista 1 y lista 2 son los números de línea de la primera sentencia de cada lista.

DATA

La instrucción DATA se puede considerar que es una instrucción de BASIC casi estándar. Se encuentra en el BASIC Microsoft y estaba en la primera versión de BASIC que se hizo (BASIC Dartmouth). El único microordenador importante que no tiene la instrucción DATA es el ZX81. Este hecho, unido a la circunstancia de que se puede pasar sin ella, es la principal razón por lo que se ha ignorado hasta ahora en este libro.

La instrucción DATA es una forma de inicializar variables a valores distintos del cero o de la cadena nula. La forma de la instrucción DATA es

DATA lista de constantes

La lista de constantes es simplemente un conjunto de valores separados por comas, que pueden ser números enteros, fracciones o incluso cadenas. Por ejemplo:

10 DATA 1,2,3,″HOLA″,—5.9

Hay que destacar que las cadenas deben estar encerradas entre comillas.

Para utilizar la instrucción DATA hay que asignar las constantes a variables. Esto se hace usando la instrucción READ. La forma de la instrucción READ es

READ lista de variables

La lista de variables es simplemente una lista de nombres de variables válidos separados por comas. Cuando el programa ejecuta una instrucción READ se pone a buscar una instrucción DATA. Si no encuentra ninguna, emite un mensaje de error. Si encuentra una, asigna la primera constante de la lista de constantes a la primera variable de la lista de variables, la segunda constante a la segunda variable y así sucesivamente hasta agotar todas las variables de la instrucción READ. Por ejemplo, si la siguiente instrucción READ estuviera en el mismo programa que el ejemplo anterior de instruccion DATA

20 READ A,B,A$,C

entonces A sería un 1, B sería 2.3, A$ sería HOLA y C sería —5.9.

Hay algunas cosas que deben recordarse a la hora de utilizar las instrucciones READ y DATA. Primero, debe haber por lo menos tantas constantes como variables. Puede haber constantes que no se usan nunca pero debe haber una constante para cada variable porque si la lista de constantes se agota antes que la lista de variables, el BASIC busca otra instrucción DATA y si no la encuentra genera un mensaje de error. Segundo, el tipo de cada constante debe encajar con el tipo de variable al que va asignada, y si no, sale un mensaje de error. En otras palabras, una variable numérica necesita una constante numérica y una variable de cadena necesita una cadena. Por último, si más tarde hay en el programa otra instrucción READ, la asignación de constantes a las variables continúa desde donde lo dejó la última instrucción READ. Es decir, no se vuelve atrás hasta el comienzo de la lista de constantes cada vez que sale una instrucción READ.

Esta última observación saca a la luz la cuestión de cómo volver

al principio de una lista de constantes. La respuesta es la instrucción RESTORE. Basta poner la instrucción RESTORE para que la siguiente instrucción READ comience su asignación empezando por la primera constante de la primera instrucción DATA del programa.

El empleo de las instrucciones READ y DATA pueden ahorrar bastante espacio de programa si se utilizan para inicializar variables, más que como listas de asignaciones. Para comprobar la certeza de esta afirmación, basta volver al programa del Código Morse dado en el Capítulo 4 y modificarlo utilizando instrucciones READ y DATA para inicializar la matriz C$.

Instrucciones específicas

Hay muchas más instrucciones BASIC que cada uno puede encontrar en el manual de su microordenador, pero casi todas ellas son específicas del mismo. No se quiere decir con esto que no son útiles. Sólo se afirma que son específicas. La mayor parte del tiempo se estará utilizando solo el conjunto de instrucciones de BASIC analizado aquí pero una cosa que no se debe hacer es suponer que con esto ya sabe BASIC. Hay que buscar y encontrar todas las peculiaridades que tiene el microordenador de que se disponga.

Si se utilizan cintas o discos para almacenar datos, habrá que averiguar cómo trata el BASIC que se emplea los ficheros de datos lo cual es una de las características del BASIC que más depende del microordenador utilizado. Del mismo modo, si se quiere aprovechar al máximo la pantalla o las posibilidades gráficas del microordenador, habrá que buscar instrucciones como PLOT o POS. En cualquier caso, el que haya llegado hasta aquí no debe tener problemas para entender su manual. Si no es así, no hay que olvidar que la informática es una ciencia experimental: adivinar y después probar.

Los intérpretes y los compiladores

En el Capítulo 1 se explicó que el BASIC es un lenguaje de alto nivel que ha de ser convertido en un lenguaje de nivel bajo

o Código Máquina antes de que el microordenador pueda obedecer las instrucciones. De hecho, hay dos formas distintas de llevar a cabo esta conversión: el BASIC puede ser «interpretado» o puede ser «compilado» lo que no es un asunto meramente académico. La forma de incorporación de un lenguaje en el microordenador tiene muchos efectos en las posibilidades y en la facilidad de empleo del lenguaje.

De estos dos métodos, el más fácil de entender es el compilador. Un compilador coge el programa escrito por el usuario y lo traduce al Código Máquina. Sólo una vez completamente traducido, pasa el programa. El empleo del compilador se realiza a través de distintas etapas. Primero hay que crear el texto del programa que el compilador traducirá. Esto suele implicar el empleo de otro programa llamado «editor» para teclear correctamente las líneas del programa en BASIC. Y una vez se tiene el texto, el compilador puede traducirlo al Código Máquina, que es el que puede ser procesado. Todas estas etapas pueden hacer que el desarrollo y depuración de un programa lleve mucho tiempo. Sin embargo hay una gran ventaja en el empleo de un compilador: una vez convertido completamente el programa en Código Máquina pasa muy rápidamente y una vez que se tiene un programa funcionando no hay necesidad de compilarlo cada vez que se utilice.

Si se trabaja en un microordenador con Microsoft u otra versión de BASIC de las mencionadas en este libro, estas etapas de edición, compilación y procesamiento no nos resultarán familiares. Cuando se trabaja con BASIC se pueden teclear líneas, editar líneas viejas y después teclear RUN y probar el programa sin ninguna necesidad de compilaciones o ediciones. La razón de ello es que la mayoría de los métodos más difundidos de operar con BASIC utilizan un intérprete. El intérprete no intenta convertir en Código Máquina todo el programa. En vez de eso pone el texto del programa en la memoria del microordenador y cuando se teclea RUN lo analiza, línea por línea, para averiguar lo que quiere decir. Aunque, en realidad, no tiene lugar ningún tipo de conversión a Código Máquina, cabe pensar que esta interpretación de cada línea es una traducción del programa, línea por línea, en oposición a lo que hace un compilador cuando traduce todo el programa antes de permitir su procesamiento.

Para poner en claro la diferencia, considérese lo que un compilador y un intérprete harían con una instrucción de BASIC GOTO

1000. El compilador traduciría todo el programa al Código Máquina y por lo tanto, en la forma final del programa, el BASIC que antes componía la línea 1000, sería un conjunto de instrucciones en Código Máquina que se sitúan en una dirección determinada de la memoria del microordenador. El comando GOTO 1000 sería traducido a la instrucción en Código Máquina equivalente de manera que transferiría el control al Código Máquina que generó la línea 1000, y cuando el microordenador encontrara esta instrucción de transferencia del control, mientras estuviera ejecutando el programa, la obedecería inmediatamente. Un intérprete, sin embargo, no pasaría ninguna parte del programa al Código Máquina. Por el contrario, al teclear el comando RUN, un intérprete examina el programa, línea por línea, y lo va ejecutando. Cuando llega a la línea GOTO 1000 analiza, en primer lugar, la palabra GOTO y trata de encontrarla en una lista de instrucciones de BASIC que conserva en memoria. Cuando descubre que es una instrucción GOTO examina los caracteres que siguen a la palabra GOTO y los conjunta en la forma de un número de línea que entonces busca en el resto del programa. En un programa largo, esta búsqueda puede tomar mucho tiempo porque es posible que tenga que examinar un considerable número de líneas antes de encontrar la línea 1000.

Está claro, por lo tanto, que los intérpretes y los compiladores se diferencian en la velocidad de ejecución de un programa. Lo que no está tan claro es, cómo los intérpretes, hacen que la escritura y comprobación de los programas sea más fácil. La situación ideal es utilizar un intérprete para desarrollar un programa y un compilador para traducirlo al Código Máquina y ejecutar el programa. Este ideal, combinación intérprete/compilador, es posible en la mayoría de los microordenadores, incluyendo todos los que funcionan con CP/M, pero el problema es que generalmente los compiladores son caros.

El BASIC, el Pascal, el FORTRAN y el ADA

El tema del BASIC avanzado nos lleva a considerar qué lenguaje habría que aprender ahora. No se puede negar que el BASIC tiene muchos defectos, no tantos como sus críticos nos quieren hacer

creer pero sí suficientes para que cualquier otro lenguaje constituya una alternativa a considerar. Hay que señalar que la mayor parte de la crítica contra el BASIC está basada en el error de creer que una versión de BASIC antigua o particularmente limitada es el BASIC actual, o está fundada en la errónea confusión de un buen o mal programa con un buen o mal lenguaje de programación. Se pueden escribir buenos programas en cualquier lenguaje y, aunque es cierto que algunos lenguajes facilitan una buena programación proporcionando mejores prestaciones, es difícil encontrar un lenguaje que sea al mismo tiempo respetable desde el punto de vista académico y útil desde el punto de vista práctico.

Uno de los mayores competidores del BASIC es el tan aclamado, desde el punto de vista académico, Pascal. Tiene mejores subrutinas y funciones que el BASIC y sus sentencias de control están pensadas para la programación estructurada (Véase Capítulo 6). Le faltan, sin embargo, el operador «elevación a potencia» o las funciones de tratamiento de cadenas y las formas comercializadas no aprovechan suficientemente las características del soporte físico (hardware) del microordenador que pueda poseer el lector. Entre las posibilidades que no considera se hallan las gráficas y el sonido, entre las más atractivas.

En vez de saltar a un lenguaje completamente distinto como es el Pascal, cabe la posibilidad de dejarse tentar por lenguajes como el COMAL que son una mezcla de Pascal y BASIC. Merece realmente la pena considerar estos lenguajes pero sin olvidar que hay versiones de BASIC, como el BASIC BBC, por ejemplo, que son casi tan avanzados como aquél pero que se siguen llamando BASIC y se utilizan mucho más.

Para aplicaciones técnicas merece la pena volver la vista atrás y recurrir a un lenguaje viejo y probado: el FORTRAN. Es un lenguaje que, aunque ha estado en vigor desde las primeras etapas de la informática, sigue siendo muy adecuado para aplicaciones técnicas. Tiene buenas prestaciones de subrutinas y funciones y soporta no sólo números complejos sino también aritmética compleja. Las versiones posteriores del FORTRAN (el FORTRAN 77, por ejemplo) son muy parecidas al BASIC, en el espíritu por lo menos, y no hay problemas para utilizar los dos.

No se debe tomar el último párrafo como una invocación de

los lenguajes antiguos. El caso es que ya ha llegado el momento de que un nuevo lenguaje informático se hiciera con algo del terreno que, en la actualidad, ocupa el BASIC como lenguaje informático más difundido. Pero a la hora de escribir este libro no se ve en el panorama informático ningún buen competidor en términos de sencillez en la utilización de prestaciones técnicas ofrecidas o disponibilidad comercial. Otro lenguaje informático más reciente, el ADA, está quizás a la espera de una oportunidad para barrer al BASIC y al FORTRAN y mandarlos a un museo, pero es más que dudoso que lo consiga.

Aunque hay muchos lenguajes buenos, el BASIC es el mejor.

Preguntas de autoevaluación

1. Escríbase una rutina que utilice el indicador de entrada para pedir al usuario su nombre y la fecha.

2. Escríbase un programa que imprima una lista de números aleatorios de tres cifras decimales.

3. Escríbase un programa para poner al descubierto la precisión del microordenador utilizado.

4. Búsquese en el manual del microordenador que se posea los comandos especiales y las características concretas de su BASIC y véase la forma de aplicarlas para mejorar o ampliar las posibilidades de los programas.

10
LOS MICROORDENADORES EN LA ELECTRONICA

En este capítulo final veremos algunas formas de utilizar los microordenadores en electrónica. En un estudio general de este tipo, cabe siempre la posibilidad de ver las cosas desde un punto de vista equivocado. En este caso concreto, sería demasiado fácil limitarse a enumerar los equipos basados en los microordenadores más caros y complicados que hay en el mercado e ignorar las innovaciones pequeñas que utilizan microordenadores personales de bajo costo.

La contribución del aficionado

La electrónica es una afición muy popular aunque no tanto como antes. La razón de este declive se halla en que los microordenadores están ocupando el lugar que una vez cubrió la electrónica como principal afición técnica. En la electrónica siempre ha habido un interés por los microordenadores. Incluso en los tiempos ya lejanos de los primeros transistores, las revistas de electrónica publicaban diseños de microordenadores sencillos. Sólo su alto precio y su gran tamaño frenaban su expansión. Pero en cuanto aparecieron en escena los microprocesadores, se convirtió en algo normal para los aficionados a la electrónica la realización de microordenadores. Los primeros microprocesadores eran dispositivos muy simples

y eran necesarias muchas horas de trabajo para hacer de ellos microordenadores que tuvieran alguna utilidad, pero estos esfuerzos se compensaban con el entretenimiento de combinar la electrónica y la informática. De hecho, el primer microordenador doméstico —el Altair— empezó siendo un proyecto de montaje publicado por una revista de electrónica norteamericana y abrió el camino a la amplia gama de microordenadores que hay en el mercado en la actualidad. Tan vasta es la gama de microordenadores que existe en el mercado, que ya no hay razón económica alguna para que cada usuario se monte su propio microordenador. Se puede comprar un modelo comercial por no mucho más del precio de los componentes necesarios. Es cierto, sin embargo, que queda todavía gente que sigue queriendo montar su propio microordenador aunque sólo sea por la satisfacción que eso produce. Hay que decir, de todas formas, que el hecho de comprarse el equipo ya montado no significa que no se esté interesado por la electrónica.

En el mundo de los radioaficionados existe la misma división: unos construyen su equipo y otros lo compran pero ambos grupos están unidos en el uso que hacen del equipo. Esta unidad no es tan clara en el mundo de los entusiastas de la electrónica porque algunos aficionados a la electrónica descubren que una vez adquirido un microordenador, su interés por la electrónica decae. El soporte lógico (software) de los microordenadores se convierte en algo tan absorbente que las ideas relativas al montaje de circuitos vienen a ser sustituidas por diseños de programas. Al fin y al cabo, el soporte lógico tiene muchas ventajas sobre el soporte físico (hardware). No plantea la necesidad de tener equipos especiales (todo gira alrededor del microordenador), es fácil y seguro hacer experimentos, es fácil hacer modificaciones y, una vez hecho algo, es relativamente barato, fabricarlo en serie para otros. Si se comparan todas estas características con los problemas que plantea la construcción del soporte físico (hardware) (soldadores, componentes defectuosos, componentes quemados, etc.), se verá que no es raro que mucha gente no vuelva la vista atrás, una vez dominados los secretos de un lenguaje de programación.

La otra cara del asunto es que algunos entusiastas de la electrónica piensan que el soporte lógico (software) es interesante pero no tanto como la electrónica, y por distintos motivos sale fortale-

cido su interés por la electrónica. Si el interés en el soporte físico de los microordenadores sigue vivo, entonces la tarea de mejorar el equipo comprado en la tienda les puede seguir dando satisfacciones. Una de las cosas que más satisfacciones produce, es conectar otros equipos electrónicos al ordenador. Esto introduce a los aficionados en el difícil campo de los interfaces. Hay una falta clara de personas suficientemente preparadas para abordar, a la vez, el soporte lógico (software) y el soporte físico (hardware) de un microordenador, y ese doble conocimiento es el que se precisa para resolver cualquier problema de conexión de equipos a microordenadores.

Por otro lado, es también posible que, una vez satisfecha la curiosidad sobre los microordenadores, comprando uno se desvanezca todo el interés que se pudiera tener en el hardware de los microordenadores y se recupere el anterior interés en otras áreas de la electrónica. De la misma forma, los profesionales de la informática pueden ver los microordenadores más como herramientas de trabajo que como un fin en sí mismos. La cuestión es: ¿qué pueden ofrecer los microordenadores a otras áreas de la electrónica?

Interconexión de equipos

Una de las áreas aparentemente más difícil en el área de la utilización de los microordenadores, es la conexión de los mismos a otros equipos, mediante lo que se ha dado en llamar interfaces. El problema está, en que se trata de un campo muy amplio. Es preciso saber mucha electrónica y mucha informática para estar en disposición de ver si hay o no alguna posibilidad de conectar un microordenador a algo. Si se carece de esta preparación es mejor dejarlo estar. Los aparatos que se han venido conectando a microordenadores, van desde el legendario control de la calefacción central hasta los robots. En un proyecto de este tipo hay generalmente tres focos de problemas. Obviamente hay que diseñar y probar la electrónica y el programa, pero existe también el problema de la interacción entre aquélla y éste. Es decir, es posible que el hardware funcione perfectamente y que el software esté bien escrito y depurado pero la forma en que el software utiliza el hardware puede ser incorrecta. Esta interacción es la razón principal por la que la

interconexión de equipos resulta tan difícil. Imagínese esta situación: se acaban de conectar las luces de una discoteca cuya instalación está controlada por microordenador y todo va bien pero se lanzan los destellos de la luz roja cuando deberían funcionar los flashes azules. ¿Es un problema de hardware o de software?

Se puede entrar en el campo de la realización de proyectos de interconexión de equipos a microordenadores, utilizando los microordenadores personales más sencillos y baratos. El ZX81, por ejemplo, tiene una gama de dispositivos adicionales de los que se puede disponer opcionalmente para simplificar la interconexión de todo tipo de equipos, conociéndose casos de conexión de: controladores de modelos de ferrocarriles a escala, de alarmas antirrobo y de un sincronizador de luz y sonido para discoteca. El que quiera entrar en el campo de este tipo de proyectos necesitará, además del microordenador, buenos conocimientos de electrónica digital, buenos conocimientos de programación, posiblemente hasta el nivel de lenguaje ensamblador —incluyendo el funcionamiento de convertidores analógico/digitales y digitales/analógicos—, y saber cómo funciona el microordenador que posee.

El análisis de circuitos electrónicos

Se acelerarían los trabajos de diseño y construcción de cualquier circuito electrónico si fuera posible describírselo al microordenador y obtener del microordenador una descripción del funcionamiento del circuito. Esta es la posibilidad que abre el ECAP. ECAP viene de Electronic Circuit Analysis Program (Programa de Análisis de Circuitos Electrónicos). Para poder hacer este análisis de circuitos electrónicos se necesita simplemente disponer del software necesario. Hay dos formas de conseguirlo: comprándolo o escribiéndolo. Decidirse a escribirlo es correr el riesgo de emplear tanto tiempo en desarrollar el programa que no llegue nunca al punto de la realización de los circuitos. El problema es que comprarlo no es una solución práctica pues yo no conozco en el mercado ningún programa ECAP bueno y barato para microordenadores.

Con todo, completar las prestaciones del ECAP, es un buen reto para los propietarios de microordenadores personales. El que se decida a escribir su propio software ECAP de tipo general, se en-

frentará a dos problemas que habrá de resolver. El primero, es la descripción del circuito que se ha de realizar al microordenador. Se puede optar por la solución tradicional de dar una lista de componentes y de conexiones pero hay un método, más ambicioso y satisfactorio, que consiste en utilizar un lápiz óptico para dibujar directamente el circuito en la pantalla. Cualquiera de los dos métodos exige buenos conocimientos de programación para conseguir resultados provechosos.

El segundo problema importante, lo plantea el cálculo de las tensiones e intensidades en los diversos puntos del circuito, cosa difícil incluso en el caso de que los únicos componentes utilizados sean resistencias. El problema está en que para calcular la corriente que circula por cualquier componente, es necesario conocer la tensión aplicada. Normalmente solo se especifica la tensión total en el circuito, es decir, la tensión de alimentación. La forma en que se distribuye a través del circuito depende de los valores de los componentes y de la forma en que están conectados. Para calcular tales valores es preciso escribir las ecuaciones básicas del circuito y luego resolverlas.

En los circuitos a base de resistencias y fuentes de alimentación las ecuaciones son siempre ecuaciones múltiples sencillas del tipo analizado en el Capítulo 8. Pero está claro que cualquier programa ECAP tendrá que tratar con componentes que no sean resistencias y las cosas empiezan a complicarse porque las ecuaciones resultantes son difíciles de resolver incluso con un microordenador.

A pesar de las dificultades técnicas que plantea el desarrollo de un programa ECAP es todo un reto. Por lo menos con lo explicado se puede comprender la dificultad que presentan. De hecho, lo que hay que conseguir es que el microordenador aplique los métodos tradicionales de análisis de circuitos lineales que se pueden encontrar en cualquier libro de texto. Un problema mucho más difícil es cómo el usuario se familiariza con un programa de este tipo, algo que no se puede aprender en los libros de texto. Hay una segunda aplicación de los programas ECAP, que plantea dificultades mucho más difíciles de solventar: los circuitos no lineales. El análisis de los circuitos no lineales con microordenadores es muy importante porque muchas veces el circuito únicamente se puede analizar con un microordenador.

Circuitos especiales

El problema de diseñar un programa ECAP de tipo general es arduo pero queda la posibilidad de hacer programas, que sirvan de ayuda para diseñar circuitos estándar. Por ejemplo, el programa presentado en el Capítulo 3 para calcular los valores de los componentes de un temporizador 555, está lejos de ser un programa ECAP de tipo general pero si lo que se quiere diseñar, es un circuito temporizador, un programa sencillo es tan bueno como un programa ECAP general. Del mismo modo, el programa de diseño del circuito de diodo Zener del Capítulo 8 es más útil en la práctica que un programa ECAP general para quien solo se ocupe del diseño de fuentes de alimentación.

Se podrían ir añadiendo a la lista programas de diseño de circuitos estándar hasta el punto de que el programa general fuera de todo punto innecesario. Lo único que hay que hacer para escribir un programa de diseño de un circuito estándar es hacerse con la hoja de datos de los circuitos integrados/diodos/transistores en cuestión y transferir al programa los cálculos que normalmente se harían a mano. Un consejo: no hay que caer en la tentación de dar demasiada información. ¡No hay que calcular parámetros del circuito que probablemente no serán necesarios! No obstante, el microordenador ofrece la posibilidad de aplicar ecuaciones de diseño que si hubiera que hacer los cálculos a mano habría que descartar. Por ejemplo, repitiendo los cálculos para una gama de valores distintos se puede suministrar información sobre la estabilidad del circuito.

Construcción de circuitos

La aplicación de los microordenadores al propio proceso de realización de circuitos, es probablemente la menos explorada y la más atractiva. A un nivel muy sencillo, se pueden aplicar a operaciones tan elementales como la decodificación de los códigos de colores de los condensadores y resistencias y el cálculo de los devanados de transformadores. Sin embargo, hay aplicaciones más interesantes. El trabajo de hacer las tablas de cableado de patillas para la soldadura o el «wire-wrapping» es una tarea lenta y abu-

rrida, tan aburrida que los aficionados y los que construyen prototipos se la quitan de encima cableando los circuitos directamente a partir del esquema del circuito. Si se dispusiera de un programa que sirviera para anotar el número de componentes del circuito, el número de patillas de cada componente y su conexión, se podría utilizar para que comprobara que no hay conexiones ilógicas —como patillas conectadas a sí mismas— y que luego imprimiera una lista de las conexiones y el orden en que hay que hacerlas. Se podría incluso utilizar el programa para anotar cada conexión según se van realizando.

Las aplicaciones de los microordenadores a otras fases de la realización de circuitos resultan un poco más difíciles y más caras de llevar a la práctica y están confinadas a la producción en gran escala. En la industria electrónica se utilizan ordenadores grandes para hacer el diseño de los circuitos y para comprobarlos. El atractivo de ambas aplicaciones es obvio. Basta imaginarse sentado frente a un microordenador, empuñando un lápiz óptico, con un esquema de una placa de circuito impreso presentada en la pantalla. Se señalan dos patillas y quedan instantáneamente (bueno, casi instantáneamente) conectadas entre sí. Cuando se ha acabado, la impresora genera una tabla de conexiones, un esquema del circuito... Los problemas inherentes a la realización de tal creación de software para un microordenador personal barato son grandes pero serían muchos los que se contentarían con algo que se aproximara a ese ideal.

El segundo campo de aplicación de los microordenadores —la localización de averías— es también difícil de poner en práctica por lo que los servicios de reparación de averías siguen apegados a los instrumentos tradicionales, medidor y osciloscopio, para localizar las averías. Incluso para comenzar a automatizar este aburrido trabajo se necesita conectar un microordenador a un conversor A/D y escribir un programa que pruebe el equipo, etapa por etapa. Hacerlo para todo un equipo resulta difícil pero si se quiere actuar en una unidad concreta, el problema no es tan grande. A un nivel ligeramente menos ambicioso, cabría la posibilidad de escribir un programa que sirviera de guía para la localización y reparación de averías en vez de realizar la propia tarea que corresponde al reparador. El programa podría hacer preguntas como ésta: ¿Está comprobado el fusible? Podría aceptar una respuesta, tratando siem-

pre de establecer dónde se hallaba el error. Hay muchos desarrollos y proyectos en el campo de la inteligencia artificial que prometen sustituir todos los conocimientos necesarios para el mantenimiento y reparación de equipos por un programa de microordenador pero habrá que esperar a ver si todo esto tiene salida en la práctica.

A un nivel mucho más modesto los microordenadores están empezando a cambiar la gama y el costo de los aparatos de medida comercializados. En principio, los microprocesadores abrieron un camino en el campo de los equipos de medida, constituyendo controladores sobre los que el usuario no tenía porque saber nada. Ahora se da una tendencia, cada vez mayor, a facilitar el conocimiento del microordenador y a animar a los usuarios a utilizarlo adecuadamente. Por ejemplo, ahora se puede comprar una placa adicional para el Apple II que lo convierte en un osciloscopio de dos canales (10 MHz), por un precio similar al de un osciloscopio estándar. Los instrumentos basados en microordenadores no son sólo más baratos sino que son mejores en el sentido de que se pueden utilizar de muchas más formas que las que permitiría el equipo tradicional. Por ejemplo, un osciloscopio basado en microordenador, se puede utilizar para lograr el promedio de señales periódicas, para obtener el seguimiento de señales no periódicas, etc.

La electrónica como soporte lógico (software)

Hasta ahora, hemos considerado el microordenador desde el punto de vista de una herramienta para la realización de nuevos equipos electrónicos. Sin embargo, también se le puede considerar como el mejor medio para crear nuevos soportes físicos (hardware). En este sentido, constituye la base idónea para todas las realizaciones electrónicas.

Considérese, a título de ejemplo, el problema de diseñar y construir un nuevo receptor de radio. Todos sabemos cómo hacerlo utilizando componentes estándar pero, ¿cómo realizarlo utilizando básicamente el microordenador? Imaginemos que disponemos de un microordenador muy rápido con un conversor A/D conectado a una antena y con un conversor D/A conectado a un altavoz. Con esto ya disponemos de todo el soporte físico (hardware) necesario para

construir un aparato de radio. El resto del montaje sería cuestión exclusiva del soporte lógico (software). La señal digitalizada de la antena sería tratada por filtrado digital, sería amplificada y la señal de audio se recuperaría simplemente con aritmética.

Después el torrente de números que representa la señal audio se entregaría al conversor D/A y luego se transmitiría al mundo exterior mediante el altavoz. Las ventajas de esta solución son claras: los cambios de diseño serán sencillamente cambios de programa, las imperfecciones introducidas por los componentes tradicionales se eliminarían y el mismo microordenador podría utilizarse para muy diferentes productos por lo que todos podrían abaratarse.

La razón de que, por el momento, esta solución no se lleve a la práctica es igualmente clara: no hay microordenadores suficientemente rápidos como para que esta realización sea factible. Cabe que este problema persista pues por muy rápidos que lleguen a ser los microordenadores siempre serán más rápidos sus componentes. Esto significa que el empleo directo de tales componentes en los equipos a utilizar ofrecerá siempre la ventaja de una mayor rapidez. No obstante, en algunas aplicaciones, esta mayor velocidad no será un factor decisivo y el microordenador irá convirtiéndose, poco a poco, en el elemento estándar para la realización de nuevos circuitos electrónicos pues ya se ha llegado al punto en que el tratamiento digital de señales de audio es tan importante como los métodos analógicos y el nuevo microordenador es la mejor base para cualquier complejo de lógica digital a menos que su funcionamiento precise velocidades extraordinariamente elevadas.

Con el paso del tiempo el papel del microordenador en la electrónica irá disminuyendo por la sencilla razón de que el microordenador se convertirá en electrónica.

RESPUESTAS A LAS PREGUNTAS DE AUTOEVALUACION

Capítulo 2

1
```
(a)17 (b)2.5 (c)8.4 (d)2.5 (e)0.
25
```

2
```
  10 PRINT "LONGITUD DEL LADO ="

  20 INPUT L
  30 LET A=L*L
  40 PRINT "AREA=";A
```

3
```
  10 PRINT "LONGITUD DEL PRIMER
LADO=";
  20 INPUT L
  24 PRINT "LONGITUD DEL SEGUNDO
 LADO=";
  26 INPUT S
  30 LET A=L*S
  40 PRINT "AREA=";A
```

4
```
  10 PRINT "INDUCTANCIA =";
  20 INPUT L
  30 PRINT "DIAMETRO DEL HILO=";
  40 INPUT D
  50 PRINT "RADIO DEL NUCLEO=";
  60 INPUT R
  70 LET N=(0.4*L*D+0.16*L*L*D*D
+0.0033*R*R*R*L)/(0.0031*R*R)
  80 PRINT "NUMERO DE ESPIRAS=";
N
```

Capítulo 3

1
```
  10 FOR I=1 TO 5 STEP 0.1
  20 LET M=I*25
  30 PRINT I;"PULGADAS=";M;" MIL
IMETROS"
  40 NEXT I
```

2

```
  1 PRINT "VALOR INICIAL=";
  2 INPUT S
  3 PRINT "VALOR FINAL";
  4 INPUT F
  5 PRINT "TAMA/O DEL PASO";
  6 INPUT C
 10 FOR I=S TO F STEP C
 20 LET M=I*25
 30 PRINT I;"PULGADAS=";M;" MIL
IMETROS"
 40 NEXT I
```

3

```
  10 PRINT "RESISTENCIA A CERO C
=";
  20 INPUT R
  30 PRINT "COEFICIENTE DE TEMPE
RATURA=";
  40 INPUT A
  50 PRINT "TEMPERATURA INICIAL=
";
  60 INPUT S
  70 PRINT "TEMPERATURA FINAL=";
  80 INPUT F
  90 FOR T=S TO F
 100 LET M=R*(1+A*T)
 110 PRINT "A ";T;" C LA RESISTE
NCIA=";M
 120 NEXT T
```

4

```
 10 INPUT R
 20 IF R>=1000 THEN GO TO 50
 30 PRINT R;"OHMIOS"
 40 GO TO 10
 50 IF R>=1000000 THEN GO TO 90
 60 LET R=R/1000
 70 PRINT R;"K OHMIOS"
 80 GO TO 10
 90 LET R=R/1000000
100 PRINT R;"M OHMIOS"
```

5

```
 110 GO TO 10
 180 FOR J=S TO F STEP (F-S)/N
 190 PRINT "PARA R=";J;"K OHMIOS
 C ES ";T/(1.1*J);"MICROFARADIOS
"
 200 NEXT J
```

Capítulo 4

1

```
10 DIM A(10)
20 FOR I=1 TO 10
30 INPUT A(I)
40 NEXT I
50 FOR I=10 TO 1 STEP -1
60 PRINT A(I)
70 NEXT I
```

2

```
 4 PRINT "CUANTOS";
 6 INPUT N
10 DIM A(N)
20 FOR I=1 TO N
30 INPUT A(I)
40 NEXT I
50 FOR I=N TO 1 STEP -1
60 PRINT A(I)
70 NEXT I
```

3

```
10 INPUT A$
20 FOR I=LEN (A$) TO 1 STEP -1
30 LET B$= MID$(A$,I,1)
40 PRINT B$;
50 NEXT I
```

o en el BASIC ZX:

```
10 INPUT A$
20 FOR I=LEN (A$) TO 1 STEP -1
30 LET B$=A$(I TO I)
40 PRINT B$;
50 NEXT I
```

4

```
  10 DIM A$(10)
  20 LET A$(1)="NEGRO"
  30 LET A$(2)="MARRON"
  40 LET A$(3)="ROJO"
  50 LET A$(4)="NARANJA"
  60 LET A$(5)="AMARILLO"
  70 LET A$(6)="VERDE"
  80 LET A$(7)="AZUL"
  90 LET A$(8)="VIOLETA"
 100 LET A$(9)="GRIS"
 110 LET A$(10)="BLANCO"
 120 PRINT "PRIMER COLOR=";
 130 INPUT C$
 140 LET C=0
 150 FOR I=1 TO 10
 160 IF C$=A$(I) THEN LET C=I
 170 NEXT I
 180 IF C=0 THEN GO TO 120
 190 LET R=C-1
 200 PRINT "SEGUNDO COLOR=";
 210 INPUT C$
 220 LET C=0
 230 FOR I=1 TO 10
 240 IF C$=A$(I) THEN LET C=I
 250 NEXT I
 260 IF C=0 THEN GO TO 200
 270 LET R=R*10+C-1
 280 PRINT "TERCER COLOR=";
 290 INPUT C$
 300 LET C=0
 310 FOR I=1 TO 10
 320 IF C$=A$(I) THEN LET C=I
 330 NEXT I
 340 IF C=0 THEN GO TO 280
 350 LET R=R*10↑(C-1)
 360 PRINT "RESISTENCIA=";R;"OHM
IOS"
 370 GO TO 120
```

Con el BASIC ZX, se efectúan cambios, quedando así las modificaciones:

—línea 10: DIM A$(10,8)
—se añade la línea 15:
DIM C$(8)

Y naturalmente, a las sentencias se les antepone LET.

Capítulo 5

1
```
  10 PRINT "INICIO=";
  20 INPUT S
  30 PRINT "FIN=";
  40 INPUT E
  50 PRINT "STEP=";
  60 INPUT C
  70 FOR I=S TO E STEP C
  80 LET A=SQR (I)
  90 PRINT "LA RAIZ CUADRADA DE"
;I;"ES";A
 100 NEXT I
```

2
```
  10 PRINT "INICIO=";
  20 INPUT S
  30 PRINT "FIN=";
  40 INPUT E
  50 PRINT "STEP=";
  60 INPUT C
  70 FOR I=S TO E STEP C
  80 LET A=SIN (I)
  85 LET B=COS (I)
  90 PRINT "ANGULO (EN RAD)=";I;
"SENO=";A;"COS=";B
 100 NEXT I
```

3
```
10 DEF FN L(X)=LOG(X)*0.434295
```

4
```
  10 DEF FN S(X)=X-X↑3/6+X↑5/120
-X↑7/5040
  20 INPUT X
  30 LET A=FN S(X)
  40 LET B=SIN (X)
  50 PRINT A,B,A-B
  60 GO TO 20
```

Como se puede comprobar, los valores de A y B se aproximan siempre que el valor de X sea pequeño. En otras palabras, FNS es una forma de calcular SIN(X). Podría ser la forma que emplea el microordenador que estamos utilizando.

Capítulo 7

1

A B C	A AND B AND C	(A AND B) OR (A AND C)	A AND (B OR C)
000	0	0	0
001	0	0	0
010	0	0	0
011	0	0	0
100	0	0	0
101	0	1	1
110	0	1	1
111	1	1	1

2

```
IF V>=L AND V<=H THEN EN ESCALA
```

3

```
 10 LET T=-1
 20 LET F=0
 30 FOR R=T TO F
 40 FOR S=T TO F
 50 FOR Q=T TO F
 60 PRINT R;"  ";S;"  ";Q;"  ";NOT
(R AND NOT (Q AND S))
 70 NEXT Q
 80 NEXT S
 90 NEXT R
```

Esta es la tabla de la verdad de un multivibrador RS. Se considera que: —1 (cierto): 0 (falso).

Capítulo 9

1

```
   10 INPUT "NOMBRE",N$
   20 INPUT "CUAL ES SU FECHA DE
NACIMIENTO",D$
   30 PRINT N$,D$
```

2

```
10 LET R=RND(0)
20 PRINT INT (R*1000)/1000
30 GO TO 10
```

(o podría haberse utilizado PRINT USING)

3

```
10 LET L=1
20 LET S=L+1
30 IF L=S THEN PRINT L
40 LET L=L*10
50 LET S=L+1
60 GO TO 30
```

El primer valor de L que se imprime es tan grande que no se puede distinguir del de L + 1. Sin embargo, hay otras formas de detectar la precisión de la versión de BASIC que se está utilizando.

OBRAS DE EDICIONES TECNICAS REDE

ALARMA ELECTRÓNICA (Libro n.º 102)
150 págs., 91 figs.

ALTA FIDELIDAD A BAJO COSTE (Libro n.º 87)
212 págs., 117 figs.

AUDIO REPARACIÓN (Libro n.º 126)
Autor: Felipe Mor. 170 págs., 187 figs.

AUTOMATISMOS DE FÁCIL CONSTRUCCIÓN (Libro n.º 107)
128 págs., 89 figs.

BIOELECTRÓNICA (Libro n.º 149)
132 págs., 72 figs.

CIRCUITOS COMPROBADOS-I:
AUDIO-1 (Libro n.º 111)
84 págs., 100 figs.

CIRCUITOS COMPROBADOS-II:
MONTAJES PRÁCTICOS (Libro n.º 115)
84 págs., 121 figs.

CIRCUITOS COMPROBADOS-III:
PRÁCTICA DIGITAL (Libro n.º 118)
80 págs., 134 figs.

CIRCUITOS COMPROBADOS-IV:
CIRCUITOS INTEGRADOS-1 (Libro n.º 128)
84 págs., 120 figs.

CIRCUITOS COMPROBADOS-V:
AUDIO-2 (Libro n.º 131)
88 págs., 100 figs.

CIRCUITOS COMPROBADOS-VI:
JUEGOS ELECTRÓNICOS-1 (Libro n.º 132)
80 págs., 100 figs.

CIRCUITOS COMPROBADOS-VII:
ANTI-ROBO (Libro n.º 135)
76 págs., 114 figs.

CIRCUITOS COMPROBADOS-VIII:
JUEGOS ELECTRÓNICOS-2 (Libro n.º 141)
80 págs., 96 figs.

CIRCUITOS COMPROBADOS-IX:
AUDIO-3 (Libro n.º 143)
82 págs., 70 figs.

CIRCUITOS COMPROBADOS-X:
TELEMANDO (Libro n.º 146)
80 págs., 118 figs.

CIRCUITOS COMPROBADOS-XI:
COMPROBADORES (Libro n.º 152)
84 págs., 111 figs.

CIRCUITOS COMPROBADOS-XII:
AUDIO-4 (Libro n.º 154)
80 págs., 118 figs.

CIRCUITOS COMPROBADOS-XIII:
ELECTRÓNICA EN EL AUTOMÓVIL (Libro n.º 158)
84 págs., 124 figs.

CIRCUITOS COMPROBADOS-XIV:
LUCES SICODÉLICAS Y JUEGOS LUMINOSOS
(Libro n.º 160)
80 págs., 100 figs.

CIRCUITOS COMPROBADOS-XV:
CIRCUITOS INTEGRADOS-2 (Libro n.º 164)
80 págs., 102 figs.

CIRCUITOS COMPROBADOS-XVI:
ELECTRONICA PARA TRENES DE JUGUETE
(Libro n.º 181)
82 págs., 102 figs.

CIRCUITOS ELECTRÓNICOS CONTROLADOS POR ORDENADOR (Libro n.º 194)
136 págs., 29 figs.

COMODIDADES ELECTRÓNICAS (Libro n.º 99)
124 págs., 75 figs.

COMUNICACIÓN INSTANTÁNEA (Libro n.º 109)
114 págs., 54 figs.

CON UN TRANSISTOR, MÚLTIPLES MONTAJES COMPROBADOS (Libro n.º 120)
118 págs., 62 figs.

CON DOS TRANSISTORES, MÚLTIPLES MONTAJES COMPROBADOS (Libro n.º 124)
140 págs., 67 figs.

CON TRES TRANSISTORES, MÚLTIPLES MONTAJES COMPROBADOS (Libro n.º 125)
132 págs., 64 figs.

CONSTRUCCIÓN DE CAJAS ACÚSTICAS (Libro n.º 138)
122 págs., 79 figs.

CONTRAESPIONAJE ELECTRÓNICO (Libro n.º 93)
115 págs., 50 figs.

CURSO RÁPIDO DE TECNOLOGÍA DIGITAL (Libro n.º 202)
Autor: Louis E. Frenzel jr. 228 págs., tamaño 28 × 22 cms.

ELECTRÓNICA AL SERVICIO DEL AUTOMOVILISTA
(Libro n.º 122)
146 págs., 68 figs.

ELECTRÓNICA EN LA FOTOGRAFÍA (Libro n.º 121)
126 págs., 52 figs.

ESPIONAJE ELECTRÓNICO (Libro n.º 84)
152 págs., 71 figs.

ESQUEMARIOS DE TV B/N
Tamaño: 27 × 21 cms.

ESQUEMARIO TV/I (libro n.º 21)
ESQUEMARIO TV/II (libro n.º 22)
ESQUEMARIO TV/III (libro n.º 55)
ESQUEMARIO TV/IV (libro n.º 67)
ESQUEMARIO TV/V (libro n.º 76)
ESQUEMARIO TV/VI (libro n.º 81)
ESQUEMARIO TV/VII (libro n.º 88)
ESQUEMARIO TV/VIII (libro n.º 94)
ESQUEMARIO TV/IX (libro n.º 100)
ESQUEMARIO TV/X (libro n.º 105)
ESQUEMARIO TV/XI (libro n.º 113)
ESQUEMARIO TV/XII (libro n.º 127)
ESQUEMARIO TV/XIII (libro n.º 140)
ESQUEMARIO TV/XIV (libro n.º 155)

ESQUEMARIO DE MAGNETÓFONOS Y CASSETTES-I
(Libro n.º 85)

ESQUEMARIO DE MAGNETÓFONOS Y CASSETTES-II
(Libro n.º 103)

ESQUEMARIO DE MAGNETÓFONOS Y CASSETTES-III
(Libro n.º 123)

ESQUEMARIO DE MAGNETÓFONOS Y CASSETTES-IV
(Libro n.º 130)

ESQUEMARIO DE MAGNETÓFONOS Y CASSETTES-V
(Libro n.º 133)

ESQUEMARIO DE MAGNETÓFONOS Y CASSETTES-VI
(Libro n.º 139)

ESQUEMARIO DE MAGNETÓFONOS Y CASSETTES-VII
(Libro n.º 144)

ESQUEMARIO DE MAGNETÓFONOS Y CASSETTES-VIII
(Libro n.º 159)

ESQUEMARIO DE TVC-I (Libro n.º 112)

ESQUEMARIO DE TVC-II (Libro n.º 114)

ESQUEMARIO DE TVC-III (Libro n.º 116)

ESQUEMARIO DE TVC-IV (Libro n.º 117)

ESQUEMARIO DE TVC-V (Libro n.º 119)

ESQUEMARIO DE TVC-VI (Libro n.º 129)

ESQUEMARIO DE TVC-VII (Libro n.º 134)

ESQUEMARIO DE TVC-VIII (Libro n.º 136)

ESQUEMARIO DE TVC-IX (Libro n.º 137)

ESQUEMARIO DE TVC-X (Libro n.º 142)

ESQUEMARIO DE TVC-XI (Libro n.º 145)

ESQUEMARIO DE TVC-XII (Libro n.º 150)

ESQUEMARIO DE TVC-XIII (Libro n.º 157)

ESQUEMARIO DE TVC-XIV (Libro n.º 161)

ESQUEMARIO DE TVC-XV (Libro n.º 162)

REPARACIÓN TV-I (Libro n.º 77)
Autor: Felipe Mor. 300 págs., 472 figs.

REPARACIÓN TV-II (Libro n.º 110)
Autor: Felipe Mor. 304 págs., 470 figs.

REPARACIÓN TV COLOR (Libro n.º 153)
140 págs., 125 figs.

ROBÓTICA PRÁCTICA (Libro n.º 200)
Autor: José M.ª Angulo. 376 págs., 263 figs.

SEGURIDAD ELECTRÓNICA (Libro n.º 108)
120 págs., 60 figs.

SOLDADURA ELÉCTRICA (Libro n.º 147)
Autor: Felipe Mor. 104 págs., 69 figs.

TIRISTOR. APLICACIONES, CARACTERÍSTICAS, FUNCIONAMIENTO (Libro n.º 73)
Autor: R. Swoboda. 104 págs., 50 figs.

TV COLOR BÁSICA (Libro n.º 166)
170 págs., 84 figs., 12 láminas color.

60.000 TRANSISTORES (Libro n.º 163)
404 páginas.

MICROINFORMÁTICA

APLICACIONES DEL CÓDIGO MÁQUINA PARA EL ZX-SPECTRUM (Libro n.º 190)
Autor: David Laine. 170 págs.

PROGRAMACIÓN EN CÓDIGO MÁQUINA PARA EL ZX-81 Y EL SPECTRUM (Libro n.º 175)
Autor: Joan Sales Roig. 158 págs.

GUÍA PRÁCTICA PARA LA PROGRAMACIÓN CREATIVA DEL SPECTRUM (Libro n.º 195)
Autor: Mike James. 302 págs.

LA MEJOR PROGRAMACIÓN DEL COMMODORE 64 POR LA PRÁCTICA (Libro n.º 193)
Autores: Peter Lupton y Frazer Robinson.
302 págs., 70 figs.

GUÍA PRÁCTICA DEL PROGRAMADOR EN CÓDIGO MÁQUINA DEL SPECTRUM (Libro n.º 197)
Autor: R. Ross-Langley. Tamaño: 27 × 21 cms. 216 págs.

INTRODUCCIÓN A LA INTELIGENCIA ARTIFICIAL CON EL SPECTRUM
Autores: Keith y Steven Brain. 188 págs.

VARIOS

TIROS DE COMBATE Y DEFENSA PERSONAL (Libro n.º 182)
Autor: Siegfried F. Hübnner. 260 págs., 308 figs.

ARMAS DE COMBATE (Libro n.º 196)
Autor: Dominique Venner. 330 págs.

ediciones técnicas



REDE

BARCELONA
(ESPAÑA)